RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Georg Klein

Entwicklung einer USB-Anbindung für einen autonomen Steuerrechner

Diplomarbeit

HD-KIP-05-03

KIRCHHOFF-INSTITUT FÜR PHYSIK

Fakultät für Physik und Astronomie

Ruprecht-Karls-Universität Heidelberg

Diplomarbeit
im Studiengang Physik

vorgelegt von

Georg Klein
aus Ludwigshafen am Rhein

1. März 2005

Entwicklung einer USB-Anbindung für einen autonomen Steuerrechner

Georg Klein

Die Diplomarbeit wurde ausgeführt am

Kirchhoff-Institut für Physik

unter der Betreuung von

Herrn Prof. Dr. Volker Lindenstruth

Kurzfassung

Clusterrechner gewinnen in der Aufnahme und Analyse wissenschaftlicher Daten zunehmend an Bedeutung. Die Wartung großer Cluster-Systeme ist jedoch zeitaufwändig und dadurch kostspielig. Für den HLT-Cluster des Beschleuniger-Experiments ALICE wird am Lehrstuhl für Technische Informatik des Kirchhoff-Instituts für Physik ein Steuerrechner in Form einer PCI-Erweiterungskarte entwickelt. Diese Karte reduziert den Verwaltungsaufwand erheblich, indem sie dem Cluster-Administrator umfangreiche Diagnose- und Fernwartungsmöglichkeiten zur Verfügung stellt. Die Karte ist dabei ein vollständig autonomes System und kann unabhängig vom Zustand des Betriebssystems des Clusterknotens arbeiten. Die Komponenten der Karte werden im Rahmen dieser Arbeit um einen USB-Baustein erweitert. Dazu wird eine PLD-basierte Hardware-Anbindung zwischen dem Prozessor der Karte und dem Schnittstellen-Baustein vorgestellt. Darauf aufbauend wird eine C-Bibliothek zur Entwicklung von Anwendungssoftware beschrieben. Diese ermöglicht dem Programmierer einen einfachen Zugriff auf die Schnittstelle. Zur Demonstration der Bibliotheksfunktionen wird eine Monitorsoftware vorgestellt. Die beschriebenen Hard- und Softwarekomponenten ermöglichen schließlich die Entwicklung von Firmware für den Schnittstellenbaustein, die für den Hostrechner einen USB-basierten Massenspeicher emuliert.

Abstract

In acquisition and analysis of scientific data, cluster-computers are gaining importance. However, maintenance of cluster systems is time-consuming and therefore costly. For the particle physics experiment ALICE's HLT-Cluster, the chair of computer science at the Kirchhoff-Institute of Physics in Heidelberg is developing a dedivated control computer. The computer is designed as a PCI-compliant extension card. This card will facilitate the maintenance of cluster nodes, by providing an instrument for diagnosis and remote controlling to cluster administrators. The controller is autonomous and entirely independent of the host computer's operating system and its status. The components of the card are extended by a USB controller chip. A PLD based interface between the card's CPU and the USB controller is introduced. A C-library for application software development is presented. The library provides application programers with easy access to the hardware interface. A monitor software is presented as an example for library usage. The introduced hard- and software components provide the means of development of firmware for the USB controller chip to emulate a USB-based mass storage device for the host computer.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Um den zunehmend komplexen Aufgabenstellungen in Industrie und Wissenschaft begegnen zu können, werden im Allgemeinen Großrechenanlagen unterschiedlicher Architektur eingesetzt. Als Alternative zu den klassischen, massiv parallelen Supercomputern, wie etwa denen vom Typ Cray T3D, haben sich *Clusterrechner* als Standard etabliert. Ein Clusterrechner besteht aus Dutzenden bis Tausenden von Standard-PCs[1] (Knoten), die über schnelle, spezialisierte Netzwerktechniken wie Myrinet oder Scalable Coherent Interconnect verbunden werden.[2] Clusterrechner stellen 294 der 500 leistungsstärksten Rechner weltweit, das entspricht fast 60%.[3] Eine Reihe von Gründen spricht für den Einsatz solcher Systeme:

- **Skalierbarkeit** Reicht die Rechenleistung für ein gegebenes Problem nicht mehr aus, können weitere Rechner hinzugefügt werden.[4]
- **Verfügbarkeit** Ausfallende Rechnerkomponenten oder ganze Knoten können in der Regel im laufenden Betrieb des Clusters ersetzt werden.
- **Kosten** Die Rechenleistung der verwendeten Standard-PCs ist in der letzten Dekade exponentiell gestiegen. Gleichzeitig sind die Preise stetig gefallen.

Diesen Vorteilen steht ein erhöhter Aufwand an Administration und technischer Wartung gegenüber. Alleine die Lokalisierung einer ausgefallenen Komponente kann bei Anlagen mit tausenden von Knoten eine nicht zu unterschätzende Herausforderung sein. Um eine hohe Verfügbarkeit des Clusters zu gewährleisten, müssen solche defekten Knoten möglichst frühzeitig erkannt werden – idealerweise bereits vor einem Totalausfall. Ein eventuell nötiger Neustart muss unkompliziert und schnell stattfinden. Im Regelfall unterstützt das jeweilige Betriebssystem des Knotens solche Wartungsaufgaben in Form von Ereignisanzeigen oder Warnmeldungen und verfügt über Fernzugriffsdienste zur Behebung des Problems. Reagiert jedoch ein Knoten nicht mehr oder tritt ein Fehler auf, bevor das Betriebssystem zur Verfügung steht, so bleibt nur der direkte Zugriff auf die defekte Hardware. Das gilt auch, wenn das *Basic Input Output System* (BIOS) eines Knotens ersetzt werden muss. Es wäre also von großem Nutzen, wenn ein vom Betriebssystem unabhängiges Medium zur Verfügung stünde, um auf fehlerhafte Knoten aus der Ferne zuzugreifen und die aufgetretenen Probleme zu beheben.

[1] Für den Massenmarkt produzierte Geräte, die mit identischer oder ähnlicher Konfiguration in großen Mengen gekauft werden können.

[2] Dies ist eine sehr freie Definition für Clusterrechner. Siehe [Pfi98] für eine umfassende Diskussion des Themas.

[3] Stand vom November 2004, Quelle: [T500].

[4] Der zu erwartende Zugewinn an Rechenleistung hängt stark von der jeweiligen Problemstellung ab. Da die zu berechnenden Daten auf die einzelnen Knoten verteilt werden müssen, ist das Verhältnis von Übertragungs- zu Rechenzeit ein begrenzendes Element. Clusterrechner sind daher besonders für Probleme geeignet, die sich gut Parallelisieren lassen.

Eine zunehmende Anzahl an Experimenten in der Teilchenphysik verwendet zur Aufnahme und Analyse der Daten Rechnercluster. Für den High-Level-Trigger-Rechnercluster des ALICE[5]-Experiments und das CBM[6]-Experiment wird am Lehrstuhl für Technische Informatik des Kirchhoff-Instituts für Physik die *Cluster Interface Agent* (CIA) Karte entwickelt, welche die oben geforderten, betriebssystemunabhängigen Fernwartungsdienste ermöglicht. Dabei handelt es sich um eine Erweiterungskarte nach dem Low-Profile Peripheral Component Interface (PCI) Standard. Sie passt in nahezu jeden aktuell verfügbaren kommerziellen Rechner (Host). Die Karte verfügt über eine eigene Fast-Ethernet-Schnittstelle und eine unabhängige Energieversorgung. Als Betriebssystem wird die quelloffene Unix-Variante Linux verwendet. Zu den Funktionen der Karte gehören:

- Protokollierung der Power-On-Self-Test (POST) Meldungen
- Erkennung der PCI-Hardware
- Zugriff auf den Host-Speicher über PCI
- Emulation einer VGA-kompatiblen Grafikkarte (Video Graphics Adapter)
- Export der Host-Grafikkartendaten über das VNC-Protokoll (Virtual Network Computing)[7]
- Drahtlose Kommunikation über eine Infrarot-Schnittstelle
- Neustart des Hostrechners

Für die erste Fassung der Karte (CIA1) wurde eine Diskettenemulation implementiert, u. a. um den Host mit neuen BIOS-Versionen zu versehen. Auch wenn die meisten Rechner noch mit Diskettenlaufwerken ausgestattet sind, gilt das Medium Diskette jedoch schon lange als nicht mehr zeitgemäß. Sowohl die speicherbaren Datenmengen als auch der Datendurchsatz sind für die meisten heutigen Anwendungsgebiete nicht mehr ausreichend. Für die zweite Version der Karte (CIA2) wurde daher beschlossen eine alternative Technik für ein austauschbares Massenspeichermedium zur Verfügung zu stellen. Die Wahl fiel auf den bewährten *Universal Serial Bus* (USB). Der USB-Standard ist etabliert und weit verbreitet und erlaubt über den Massenspeicher hinaus weitere Anwendungen, z. B. eine Maus- oder Tastaturemulation. Konkret wurde für die CIA2-Karte der USB-Chip CY7C67300 (EZ-Host) der Firma Cypress in das CIA-System integriert.

Die hard- und softwareseitige Anbindung des EZ-Host-Chips an den Hauptprozessor der CIA2-Karte sowie die Referenzimplementierung eines Massenspeichers sind das Thema dieser Diplomarbeit.

Nach einer einleitenden Erläuterung der wichtigsten Konzepte des USB in Kapitel 2 folgt zunächst ein Überblick über die verwendete Hardware in Kapitel 3. In Kapitel 4 werden

[5] Ein Akronym für „A Large Ion Collider Experiment“, ein Teilchenbeschleunigerexperiment, das ab 2007 am Large Hadron Collider des Europäischen Kernforschungszentrums CERN durchgeführt werden wird.

[6] Das Compressed Baryonic Matter (CBM)-Experiment ist derzeit in Planung und wird am Beschleuniger der Gesellschaft für Schwerionenforschung in Darmstadt durchgeführt.

[7] Das von der Firma AT&T entwickelte VNC-Protokoll ermöglicht es, den Bildschirminhalt eines Rechners (Server) über ein Netzwerk auf einen anderen Rechner (Viewer) zu übertragen. Der Viewer seinerseits kann Maus- und Tastaturdaten an den Server übertragen und diesen damit fernbedienen.

die verfolgten Ansätze zur hardwareseitigen Anbindung des EZ-Host-Chips beschrieben, in Kapitel 5 die darauf aufsetzende Softwareanbindung. Die für den Mikroprozessorkern des EZ-Host geschriebene *Firmware* ist in Kapitel 6 dokumentiert. Abschließend enthält Kapitel 7 eine Zusammenfassung des Entwicklungsstandes sowie einen Ausblick auf künftige Aufgaben und Erweiterungsmöglichkeiten.

2 USB-Grundlagen

In diesem Kapitel werden die Begriffe und Konzepte erklärt, die zum Verständnis der Spezifikationen für einen Massenspeicher auf USB-Basis notwendig sind. Für eine vollständige Erläuterung des USB-Protokolls sei auf die offiziellen Spezifikationen in [USB20] verwiesen.

2.1 Geschichte

Der im Januar 1996 erstmals vorgestellte Universal Serial Bus (USB) hat sich mittlerweile als Standard für den Anschluss von Peripheriegeräten durchgesetzt. Ursprünglich zur Verbindung von Rechner und Telephon und als vereinheitlichender Nachfolger für zahlreiche andere Schnittstellen[1] konzipiert, wurde der USB von der PC-Industrie zunächst schlecht akzeptiert und kaum eingesetzt. Dies änderte sich mit der Revision 1.1 der Spezifikation, die im September 1998 veröffentlicht wurde. In zunehmendem Maße wurden USB-konforme Peripheriegeräte hergestellt, was zu einer breiten Akzeptanz bei Benutzern und Herstellern führte. Die im April 2000 vorgestellte Revision 2.0 spezifizierte Erweiterungen zu deutlich höheren Bandbreiten und führte damit zum endgültigen Durchbruch. Mindestens eine, meist aber zwei bis vier USB-Schnittstellen sind mittlerweile in jedem neuen Rechner enthalten. Im Oktober 2000 wurde die Spezifikation um eine weitere Stecker/Buchsen-Kombination, Mini-B, ergänzt. Die kleinere Bauform wurde der höheren Integrationsdichte in neueren Peripheriegeräten gerecht. Als weitere Ergänzung wurde im Dezember 2001 die On-The-Go-Erweiterung (OTG)[2] herausgegeben. Sie beschreibt die Kommunikation zwischen Geräten ohne einen als Host zwischengeschalteten Rechner. So kann beispielsweise ein Photodrucker direkt an eine Kamera angeschlossen werden. Die Hostfunktionalität wird in diesem Fall von der Kamera übernommen. Zudem wurden in dieser Erweiterung ein weiterer Stecker, Mini-A, und zwei neue Buchsen, Mini-A und Mini-AB, eingeführt. Buchsen vom Typ Mini-AB können sowohl Mini-A als auch Mini-B Stecker aufnehmen und erlauben es, ein Gerät sowohl als OTG-Host als auch als Peripheriegerät zu betreiben. Der jeweilige Betriebsmodus wird dabei an einem eigenen ID-Pin im jeweiligen Stecker erkannt.

2.2 Protokoll

USB verwendet ein Master/Slave-Protokoll. Alle Kommunikation geht stets vom Master aus, dem *Host Controller*. Der Slave hat keine Möglichkeit einen Transfer zu initiieren, sondern muss auf Anfragen vom Host warten. Folgerichtig werden in der USB-Terminologie vom Host gesendete Daten als Downstream oder Outgoing bezeichnet, vom Slave als Antwort gesendete Daten als Upstream oder Ingoing. Die Spezifikation trennt Geräte in zwei

[1] Beispielsweise die RS-232 (serielle) oder IEEE-1284 (parallele) Schnittstelle.
[2] Diese wurde am 24. Juli 2003 zur aktuellen Version 1.0a revidiert.

Klassen, *Hubs* und *Functions*. Über Hubs können bis zu 127 Geräte in einer baumförmigen Topologie verbunden werden.[3] Functions erweitern das Hostsystem um Funktionalitäten, meist als Ein- oder Ausgabegerät. In einem mehrschichtigen Modell werden dabei Kommunikationskanäle, *Pipes*, zwischen Datenquellen oder -senken in Anwendungssoftware und spezifischen Komponenten[4] des Gerätes, den *Endpunkten* (EP), aufgebaut. Die Schichten des Modells werden in Abbildung 2.1 schematisch dargestellt. Auf der linken Seite ist der

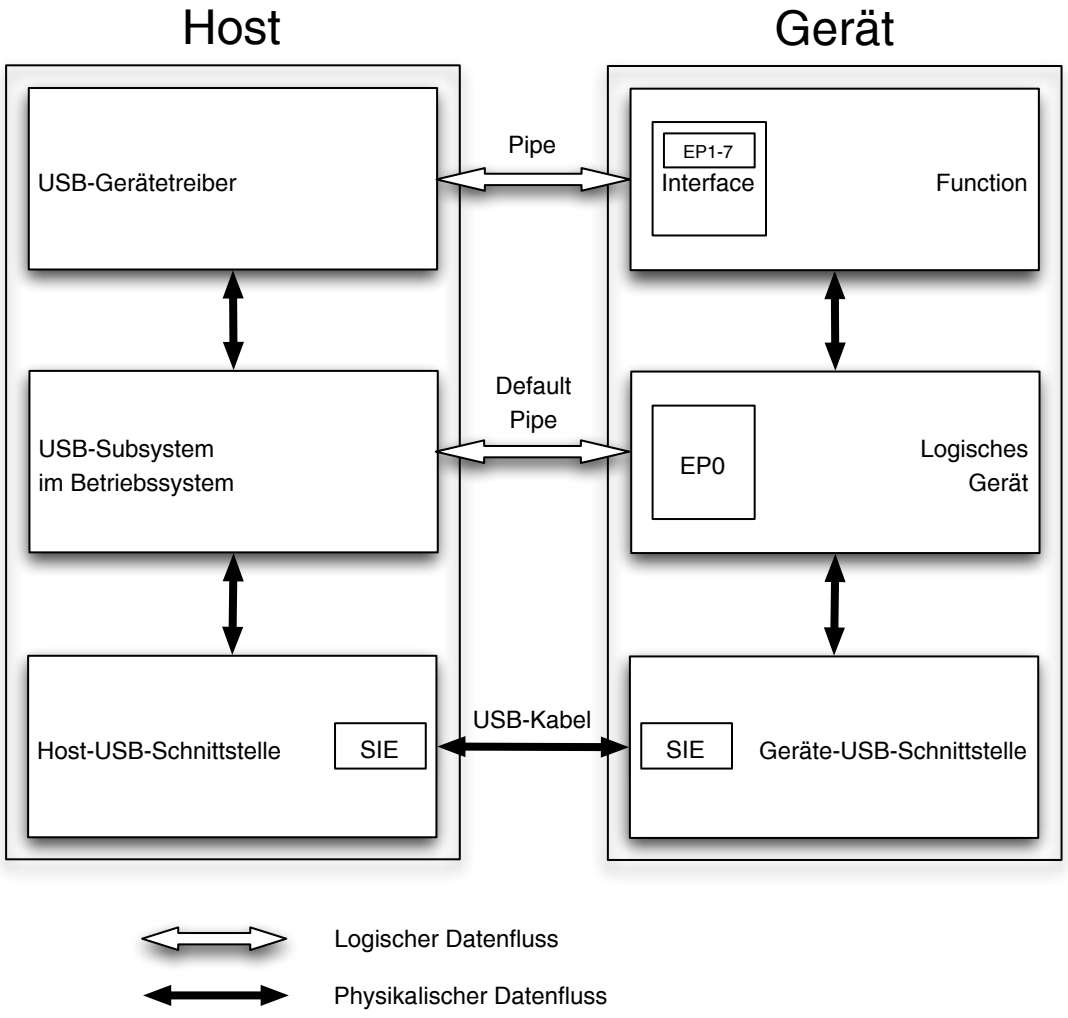


Abbildung 2.1: Schichten des USB-Kommunikationsmodells, nach [USB20]

Host dargestellt, rechter Hand das Peripheriegerät. Der physikalische Datenfluss geht von der Treibersoftware durch das Betriebssystem in die Hardware-Schnittstelle des Hosts. Von dort werden die Daten über ein USB-Kabel an das Gerät übertragen. Die Signalcodierung

[3] Ein Hub zählt dabei als Gerät, belegt also eine der 127 erlaubten Gerätenummern.
[4] Etwa eine Maustaste oder ein Mikrophon.

und Decodierung wird von den *Serial-Interface-Engines* (SIE) erzeugt. Im Gerät werden die Daten dann bis zu dem adressierten Endpunkt weitergeleitet.

Für ein Gerät können mehrere Konfigurationen definiert werden. Mit diesen können die Eigenschaften des Gerätes gezielt aktiviert oder deaktiviert werden. Hat etwa eine Videokamera eine Kamera und ein Mikrophon, so kann der Host über Wahl der Konfiguration entscheiden, ob nur die Kamera, nur das Mikrophon oder beide aktiv sein sollen. Jede Konfiguration besitzt mindestens eine Schnittstelle. Eine Schnittstelle repräsentiert Gerätekomponenten, im obigen Beispiel z. B. die Kamera. Kann eine Komponente in verschiedenen Modi betrieben werden, können alternative Schnittstellen definiert werden. Im Beispiel der Kamera könnte zwischen Farb- und Schwarz/Weiß-Betrieb gewechselt werden. USB-Gerätetreiber setzen auf Schnittstellen-Ebene an. Ein physisches Gerät kann so von mehreren Treibern benutzt werden. Ist für eine der Schnittstellen kein Treiber im Host-Betriebssystem verankert, so können die anderen Schnittstellen trotzdem verwendet werden.

Eine Komponente kann mehrere Datenquellen oder -senken besitzen. Diese werden durch die Endpunkte verfügbar gemacht. Pro Schnittstelle können bis zu sieben Endpunkte definiert werden. Dabei werden vier Varianten unterschieden:

Control-EPs werden zur Aushandlung und Steuerung der Datenkommunikation verwendet. Es existiert in jedem Gerät mindestens ein Control-Endpunkt, der *EP0*[5].

Interrupt-EPs werden für möglichst echtzeitnahe Kommunikation eingesetzt. Ein Beispiel für eine Anwendung ist eine Maus. Es ist wichtig hier festzuhalten, dass Peripheriegeräte nicht in der Lage sind, von sich aus Unterbrechungen auszulösen. Es muss zunächst eine Anfrage vom Host erfolgen, die das Gerät dann über einen Interrupt-EP beantwortet.

Bulk-EPs werden zum Transport von großen, zusammenhängenden Datenblöcken verwendet. Bulk-EPs verwenden Fehlererkennungstechniken, um die Datenintegrität zu gewährleisten. Diese Eigenschaft macht sie besonders geeignet für die Verwendung in Massenspeichern.

Isochronous-EPs werden zur Übertragung von kleinen bis mittelgroßen, synchronen Datenpaketen benutzt. Die garantierte und konstante Datenrate sowie die niedrige Latenz ist für Audio-/Video-Anwendungen, bei denen die Kontinuität der Daten besonders wichtig ist, von besonderer Bedeutung.

Die Kanäle sind, mit Ausnahme des obligatorisch vorhandenen EP0, immer unidirektional. Ihre Richtung wird über die zugehörigen Endpunkt-Deskriptoren festgelegt.[6] Soll also ein Gerät eine bidirektionale Kommunikation ermöglichen, so müssen mindestens zwei zusätzliche EPs definiert werden. Dabei können selbstverständlich auch unterschiedliche Typen parallel verwendet werden. Für Multifunktionsgeräte, wie etwa eine Tastatur mit eingebautem Mikrophon, würde man beispielsweise mindestens einen Interrupt- und einen Isochronous-EP einsetzen.

[5] In der Literatur auch etwas irreführend als *Default Pipe* bezeichnet.

[6] S. Abschnitt 2.6 zur Erklärung des Deskriptor-Konzepts sowie Tabelle A.4 für die in der vorliegenden Implementierung verwendeten Endpunkt-Deskriptoren.

2.3 Stecker und Buchsen

Man unterscheidet zwischen Steckern vom Typ A und Typ B.[7] Hosts dürfen lediglich Buchsen vom Typ A aufweisen. Peripheriegeräte dürfen nur Stecker vom Typ A als Ausgang haben. Erlaubt sind dabei fest mit dem Gerät verbundene Kabel oder Buchsen vom Typ B am Gerät. Im letzteren Fall werden Verbindungskabel eingesetzt, die auf der dem Host zugewandten Seite einen Stecker vom Typ A haben und auf der dem Gerät zugewandten Seite einen Stecker vom Typ B. Dies sind die einzigen spezifikationskonformen Kabel.[8] Eine Sonderrolle nehmen die für die Baumhierarchie notwendigen Hubs ein. Sie haben mehrere Buchsen vom Typ A, in die Peripheriegeräte eingesteckt werden, jedoch immer nur eine Buchse vom Typ B. Diese verbindet den Hub mit dem Host oder einem weiteren Hub, bis die maximale Kabellänge erreicht ist. Die mit Revision 2.0 der USB-Spezifikation sowie den Ergänzungen eingeführten Mini-Steckervarianten werden analog verwendet. Sie besitzen allerdings einen zusätzlichen Identifikations-Pin. Die Mini-AB-Buchsen sind für Mini-A- und Mini-B-Stecker geeignet und erkennen über diesen Pin, welche Steckervariante gerade eingesteckt ist.

2.4 Übertragungstechnik

Der USB verwendet ein differentielles Datensignal. Der logische Wert eines übertragenen Bits bestimmt sich aus der Differenz der beiden Datenleitungen D+ und D-. Die differentielle 1 ist definiert als $(D+) - (D-) > 200mV$ und $D+ > 2{,}0V$. Die differentielle 0 ist definiert als $(D-) - (D+) > 200mV$ und $(D-) > 2{,}0V$. Insgesamt hat ein USB-konformes Kabel vier Leitungen: Masse, D-, D+ und V_{BUS}. V_{BUS} stellt den angeschlossenen Geräten Betriebsspannung zur Verfügung, diese beträgt 5 Volt. Die Datenleitungen sind als Twisted-Pair ausgelegt, verwenden also verdrillte Datenleitungen, um das Signal-/Rauschverhältnis bei hohen Übertragungsfrequenzen zu optimieren. Die Kabellänge ist durch Signaldämpfung und -laufzeit beschränkt. Die maximal erlaubten, frequenzabhängigen Dämpfungswerte[9] sind aber so festgelegt, dass Kabel guter Qualität ca. 5 m lang sein können.

2.5 Betriebsmodi

Der USB wurde entworfen, um Peripheriegeräte in einfacher Weise anzubinden. Den unterschiedlichen Anforderungen an Datendurchsatz und Leistungsaufnahme für verschiedene Gerätegruppen[10] wurde man durch die Einführung zweier Betriebsmodi, *Low Speed* und *Full Speed*, gerecht. Mit der Revision 2.0 der USB-Spezifikation kam ein dritter Modus, *High Speed*, hinzu. Tabelle 2.1 vergleicht die drei in USB 2.0 zur Verfügung stehenden Betriebsmodi.

[7] Abbildung B.3 zeigt eine Schemazeichnung dieser Steckertypen.
[8] Der Handel bietet auch USB-Kabel vom Typ A-A oder B-B an. Diese sind aber streng genommen nicht USB-konform.
[9] Die exakten Grenzwerte sind in [USB20] detailliert beschrieben.
[10] Eine Maus z. B. stellt deutlich geringere Anforderungen als ein Scanner oder Drucker.

	Low Speed	**Full Speed**	**High Speed**
Datendurchsatz	1,5 MBit/s	12 MBit/s	480 MBit/s
Beispiele für Anwendungen	Einfache und kostengünstige Peripheriegeräte mit geringem Datendurchsatz und Energiebedarf – Maus, Tastatur, Messfühler	Geräte mit erhöhtem Datendurchsatz – Scanner, Flashspeicher, Digitalkameras	Geräte mit hohem Datendurchsatz und eigener Energieversorgung – hochauflösende Videokameras, externe Festplatten, Hochleistungsdrucker

Tabelle 2.1: Vergleich der USB-Betriebsmodi

2.6 Deskriptoren

Um dem Host Informationen über die Eigenschaften des Peripheriegerätes zu übermitteln, werden so genannte Deskriptoren eingesetzt. Dabei handelt es sich um Datenblöcke von definierter Länge und Struktur. Sie werden in der Firmware des Gerätes definiert und können vom Host abgefragt werden. Es gibt verbindliche und optionale Deskriptoren. Die ersten zwei Felder eines Deskriptors bestehen immer aus einem Byte das die Länge des Deskriptors enthält, gefolgt von einem weiteren Byte dessen Wert[11] die Art des Deskriptors anzeigt. Die folgenden Datenfelder beinhalten die für den Deskriptor relevanten Attribute des Gerätes. Die (verbindlichen) USB-Standarddeskriptoren bilden eine Hierarchie, bestehend aus:

Geräte-Deskriptor Es existiert nur ein Geräte-Deskriptor pro Gerät. Dieser enthält allgemeine Informationen zum Gerät: unterstützte USB-Version, Geräteklasse, -subklasse und -protokoll, die maximale Datenblockgröße des Control-EPs. Eine eindeutige Hersteller- und eine Produkt-Identifikationsnummer (ID) sowie eine vom Hersteller vergebene Seriennummer der Gerätes. Das letzte Feld gibt die Anzahl der für das Gerät definierten Konfigurationen an.

Konfigurations-Deskriptor(en) Jeder Konfigurations-Deskriptor beschreibt eine Konfiguration des Gerätes. Ein USB-konformes Gerät hat mindestens einen Konfigurations-Deskriptor. Er enthält eine eindeutige Kennnummer sowie die Anzahl der zur Konfiguration gehörenden Schnittstellen-Deskriptoren. Besitzt das Gerät eine eigene Energieversorgung, so bestimmt der Deskriptor, ob die Konfiguration diese verwendet. Ansonsten wird im Deskriptor festgelegt, wieviel Strom das Gerät aus dem Bus beziehen wird.[12] Zu jedem Konfigurations-Deskriptor gehören ein oder mehrere Schnittstellen-Deskriptoren.

Schnittstellen-Deskriptor(en) Jeder Schnittstellen-Deskriptor enthält eine innerhalb der Konfiguration eindeutige Kennnummer sowie die Nummer eines alternativ verwendbaren Schnittstellen-Deskriptors. Existiert nur ein Schnittstellen-Deskriptor, so sind

[11] Die erlaubten Werte sind in der USB-Spezifikation festgelegt.
[12] Die USB-Spezifikation legt dabei ein Maximum von 500 mA fest.

Kennnummer und Alternativnummer identisch. Die ebenfalls anzugebende Anzahl der Endpunkte kann 0 sein, in diesem Fall wird nur der EP0 verwendet. Wurden im Geräte-Deskriptor für Geräteklasse, -subklasse und -protokoll jeweils der Wert 0 angegeben, so werden diese Informationen in den Feldern Schnittstellenklasse, -subklasse und -protokoll im Schnittstellen-Deskriptor definiert. Auf diese Weise kann ein Gerät abhängig vom gewählten Schnittstellen-Deskriptor als anderer Gerätetyp erkannt werden.

Endpunkt-Deskriptor(en) Die Endpunkt-Deskriptoren beschreiben die Eigenschaften einzelner Gerätekomponenten. Hier werden EP-Typ und Datenflussrichtung festgelegt.

Zusätzlich sind optionale, erläuternde Zeichenketten-Deskriptoren erlaubt. Diese enthalten beispielsweise Produkt- und Herstellernamen.[13] Abschnitt A.4 enthält den für die Firmware des EZ-Host verwendeten Deskriptorensatz.

2.7 Requests

Das USB-Protokoll stellt dem Host zur Steuerung des Gerätes so genannte *Requests* zur Verfügung, z. B. um die Deskriptoren eines angeschlossenen Gerätes abzufragen oder eine der vorhandenen Konfigurationen auszuwählen. Requests werden vom Host in einer Setup-Phase als Datenpakete an den Control-EP geschickt. Tabelle 2.2 zeigt den Aufbau eines solchen Datenpakets.

Offset	Feldname	Größe	Beschreibung
0	bmRequestType	1	Request-Eigenschaften: Bit 7 bestimmt die Datenflussrichtung: 0: Host an Gerät 1: Gerät an Host Bits 6 – 5 legen den Requesttyp fest: 0: Standard 1: Klasse 2: Herstellerspezifisch 3: Reserviert
			Bits 4 – 0 beschreiben den Empfänger: 0: Gerät 1: Schnittstelle 2: Endpunkt 3: Andere 4 – 31: Reserviert
1	bRequest	1	Bezeichnung der Anfrage. Der hier übertragene Wert ändert seine Bedeutung je nach Typ des Requests.

[13] Da das Gerät über die im Geräte-Deskriptor abgelegten Hersteller- und Produkt-Identifikationsnummern eindeutig identifiziert ist, sind solche Zeichenketten-Deskriptoren nicht verbindlich.

Offset	Feldname	Größe	Beschreibung
2	wValue	2	Datenwort. Kann je nach Requesttyp unterschiedliche Bedeutung annehmen.
4	wIndex	2	Datenwort variabler Bedeutung. Wird von den meisten Requesttypen als Index oder Offset verwendet.
6	wLength	2	Bestimmt die Anzahl der Bytes in der folgenden (optionalen) Datenphase. Die USB-Spezifikation schreibt vor, dass ein Gerät nicht mehr als die hier angegebene Anzahl an Bytes übertragen darf, selbst wenn mehr Daten vorliegen sollten. Sind weniger Daten als angefordert vorhanden, so darf das Gerät auch weniger Daten liefern.

Tabelle 2.2: Aufbau eines Request-Datenpakets, Quelle: [USB20]

Requests können zusätzliche Nutzdaten erfordern. Sie werden in einer zweiten Phase, der Datenphase, übertragen. Diese umfasst die im Feld wLength angegebene Anzahl an Bytes. Ist diese null, so schließt sich keine Datenphase an, der Request wird dann mit einer Status-Phase beendet. Als Beispiel wird der *GetDescriptor*-Standardrequest angeführt[14]:

bmRequestType 10000000 (Binär). Der Host erwartet Daten vom Gerät. Es handelt sich um einen Standardrequest, Empfänger ist das Gerät.

bRequest GET_DESCRIPTOR (0x06). Mit diesem Request fordert der Host einen (im folgenden Feld näher definierten) Deskriptor vom Gerät an.

wValue Deskriptortyp und -index. Das obere Byte dieses 16-Bit-Wertes enthält die Kennung des angeforderten Deskriptors, etwa DEVICE (0x01). Das untere Byte enthält den Deskriptorindex. Dieser bestimmt bei mehreren definierten Deskriptoren identischen Typs (z. B. Zeichenketten-Deskriptoren) den Index des vom Gerät zu übertragenden Deskriptors.

wIndex Enthält den Wert Null oder eine Sprach-Identifikationsnummer (ID). USB-Deskriptoren können in mehreren Sprachen vorliegen. In diesem Fall kann über die Sprach-ID die gewünschte Deskriptorfassung angefordert werden.

wLength Deskriptorlänge. Variiert nach Typ des angeforderten Deskriptors, ein Geräte-Deskriptor z. B. hat die Länge 0x12.

Daten Die ersten <wLength> Bytes des vom Gerät angeforderten Deskriptors.

Neben den Standard-Requests, die von allen USB-konformen Geräten implementiert werden müssen, existieren auch geräteklassenspezifische Requests, Class-Requests. Diese werden in den jeweiligen Klassenspezifikationen beschrieben und müssen von allen Geräten der

[14]Für eine detaillierte Beschreibung der übrigen Standardrequests s. [USB20, S. 250ff].

jeweiligen Klasse implementiert werden. Zusätzlich kann ein Hersteller für seine Geräte eigene Requests, Vendor-Requests, definieren. Der eingesetzte USB-Chip unterstützt z. B. einen Vendor-Request, um eine neue Revision der Firmware in das EEPROM des Chips zu laden.

3 Verwendete Hardware

In diesem Kapitel wird ein kurzer Überblick über die in der Arbeit verwendeten Hardware-Komponenten gegeben. Da sich die CIA2-Karte zu Beginn der Arbeit noch in der Entwicklung befand, wurde zunächst die von der Firma Altera vertriebene EPXA1-Entwicklerplatine verwendet. Nach Fertigstellung der ersten CIA2-Musterexemplare wurde die Arbeit auf der CIA2-Plattform fortgesetzt.

3.1 Altera Excalibur Embedded Processor

Herz der CIA2-Karte ist der Excalibur-Baustein der Firma Altera. Die Excalibur-Baureihe integriert einen ARM922T-Prozessorkern, internen Speicher in Static Random Access Memory-Technik (SRAM) und einen programmierbaren Logikbaustein (PLD) aus der APEX-20KE-Baureihe. Die verwendete Excalibur-Variante, der EPXA1, verfügt über die in Tabelle 3.1 aufgeführten Eckdaten.

Eigenschaft	**Kapazität**
Systemgatter	263000
Logikzellen	4160
Eingebettete Systemblöcke	26
Speicherbits	53248
Makrozellen	416
Frei nutzbare Ein-/Ausgabe-Pins	186
Taktung des Prozessors	100 MHz
SRAM	32 kByte
Dual-Ported-SRAM	16 kByte

Tabelle 3.1: Eigenschaften des EPXA1-Prozessors, Quelle: [EHRM]

3.2 Cypress EZ-Host USB-Chip

Der im CIA2-System verwendete USB-Schnittstellenbaustein ist der von der Firma Cypress entwickelte CY7C67300 (EZ-Host). Der Chip besteht im Wesentlichen aus einem CY16-Mikroprozessorkern, der mit den für den Full-Speed-Modus notwendigen 48 MHz betrieben wird, und zwei getrennt steuerbaren SIEs. Jede dieser SIEs kann entweder zwei USB-Host- oder eine USB-Peripherieschnittstelle steuern, der jeweilige Konfigurationsmodus wird durch die auf dem Prozessor laufende Software (Firmware) gesetzt. Zusätzlich

erfüllt der EZ-Host die OTG-Spezifikationen, kann also als OTG-kompatibles Peripheriegerät verwendet werden. Eine Vielzahl von Kommunikationsschnittstellen wird unterstützt, darunter die für die vorliegende Anbindung verwendete *Host-Port-Schnittstelle* (HPI), sowie eine für die Suche nach Softwarefehlern (Debugging) hilfreiche serielle Schnittstelle. Alle Schnittstellen teilen sich die 32 vorhandenen Mehrzweck-Ein-/Ausgabe-Pins (GPIO). Die Konfiguration kann beim Initialisieren des Chips über die obersten 2 GPIOs gewählt werden. Die Konfigurationsmöglichkeiten zeigt Tabelle 3.2. Bis zu 512 kByte externer Speicher kann in Form von SRAM oder Read-Only-Memory (ROM) über die Adressleitungen A0 bis A18 angebunden werden. Dabei sind 8-Bit- und 16-Bit-Anordnungen erlaubt. Auf dem Chip selbst stehen 8 kByte RAM sowie 4 kByte ROM, jeweils in 16-Bit-Anordnung, zur Verfügung. Im ROM ist das BIOS des Chips gespeichert.

GPIO[31]	GPIO[30]	Modus
0	0	Host-Port-Interface (HPI)
0	1	High-Speed-Serial (HSS)
1	0	Serial-Peripheral-Interface (SPI)
1	1	IIC-EEPROM[1]

Tabelle 3.2: EZ-Host Boot-Konfiguration, Quelle: [CEDS]

3.3 EPXA1-Entwicklerplatine

Die Firma Altera bietet als Entwicklungshilfe für Geräte, die einen Excalibur-Baustein enthalten sollen, eine Entwicklerplatine an. Sie verfügt über einen EPXA1, dessen interner Prozessor mit 100 MHz getaktet ist. Dem Prozessor stehen 32 MByte Hauptspeicher (SDRAM) und zwei 4 MByte große Flashbausteine als Festspeicher zur Verfügung. Verbindungen zu anderen Rechnern können über eine Fast-Ethernet-Schnittstelle und zwei RS-232-Ports sowie eine JTAG[2]-Schnittstelle aufgebaut werden. Der Entwickler kann bis zu zehn Leuchtdioden und vier Drucktaster frei ansteuern. Für die Arbeit maßgeblich waren jedoch die Erweiterungs-Steckleisten. Die Pins des EPXA1 sind mit diesen Steckleisten verbunden, sie können zum Anschluss externer Komponenten genutzt werden. Zusätzlich werden Spannungsversorgung und Oszillatorsignale auf einige der Pins geleitet. Die Platine wird mit einem 25-MHz-Oszillator ausgeliefert, kann jedoch auch mit einem externen Taktgeber betrieben werden. Die wesentlichen Bausteine der Entwicklerplatine – Prozessor, Arbeitsspeicher, Flashspeicher, Ethernetschnittstelle – sind identisch oder baugleich mit den für die CIA2-Karte verwendeten.

[1] Inter-Integrated Circuit (IIC) Electically-Erasable Programmable Read Only Memory (EEPROM). Der von der Firma Philips entwickelte serielle IIC-Bus erlaubt es, mehrere Chips über einen gemeinsamen Datenbus anzusprechen. Beim EEPROM handelt es sich um einen wiederbeschreibbaren, extern angeschlossenen Festspeicher, von dem der EZ-Host seine Firmware laden könnte. Ein solcher Chip ist in das bestehende CIA-System jedoch nicht eingebunden.

[2] Der von der Joint Test Action Group festgelegte Schnittstellenstandard trägt offiziell die Bezeichnung IEEE 1149.1, wird aber meist nach dem spezifizierenden Gremium JTAG genannt.

Abbildung 3.1 zeigt die Entwicklerplatine. Der EPXA1 ist in der Mitte platziert. Fast-Ethernet- und RS-232-Schnittstellen sind oben bzw. oben rechts zu erkennen. Die Erweiterungs-Steckleisten belegen die linke Seite der Platine. Die mittleren drei Leisten werden verwendet, um eine speziell angefertigte Zusatzplatine (Mezzanine-Karte) mit dem EZ-Host-Chip an den EPXA1 anzuschließen.[3]

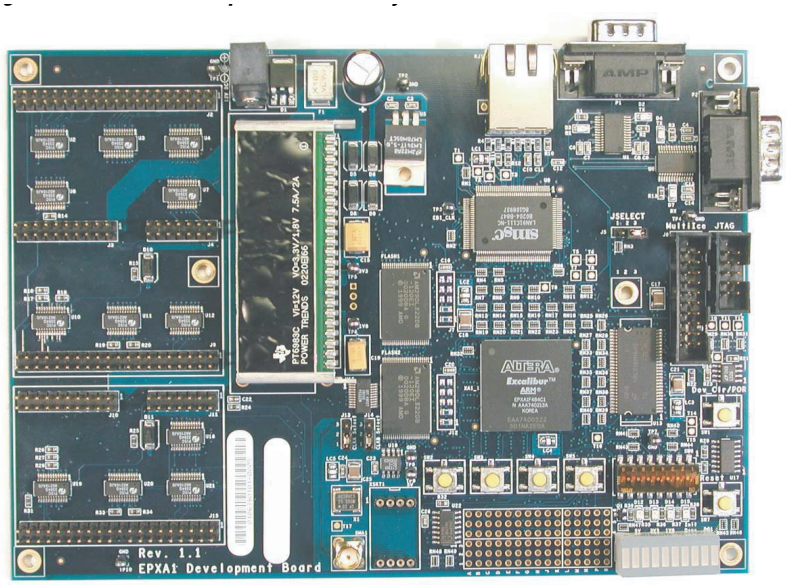


Abbildung 3.1: Photo der Altera EPXA1-Entwicklerplatine, entnommen aus [EHRM]

3.4 Mezzanine-Karte

Um bereits während der Entwicklung der CIA2-Karte an der Anbindung des USB-Chips arbeiten zu können, wurde eigens eine Mezzanine-Karte für die Erweiterungs-Steckleisten der EPXA1-Entwicklerplatine gebaut. Ein möglichst unkomplizierter Übergang zur CIA2-Karte wurde durch Verwendung der EPXA1-Pinbelegung aus den CIA2-Schaltplänen gewährleistet. Abbildung 3.2 zeigt eine Photographie der Karte. Angefertigt wurde sie in der hauseigenen Elektronikwerkstatt. Insbesondere der einfache Zugang zu den Leiterbahnen vereinfachte die Signalmessungen in der Anfangsphase erheblich. Die Karte verfügt, wie die CIA2-Karte, über zwei USB-Buchsen, eine vom USB-A-Typ für Host- und eine vom USB-B-Typ für Peripheriefunktionalität.

[3] Eine ausführliche Beschreibung der Komponenten und Pinbelegungen der Platine ist in [EHRM] zu finden.

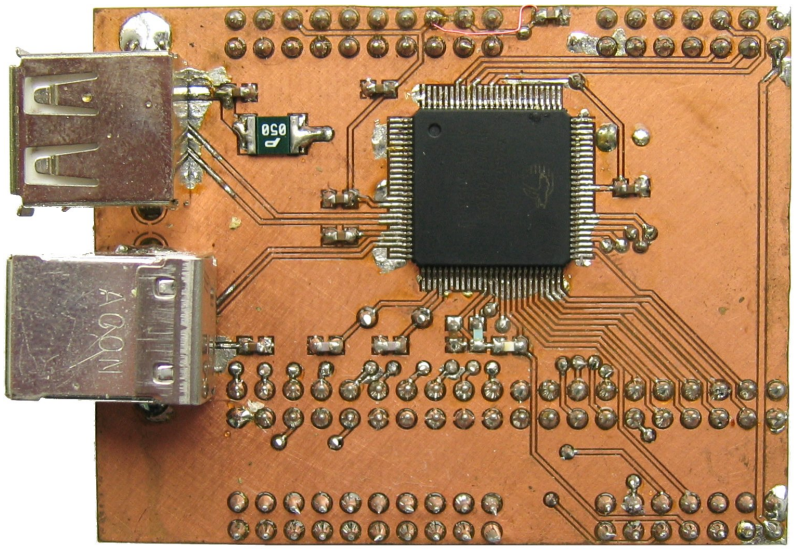
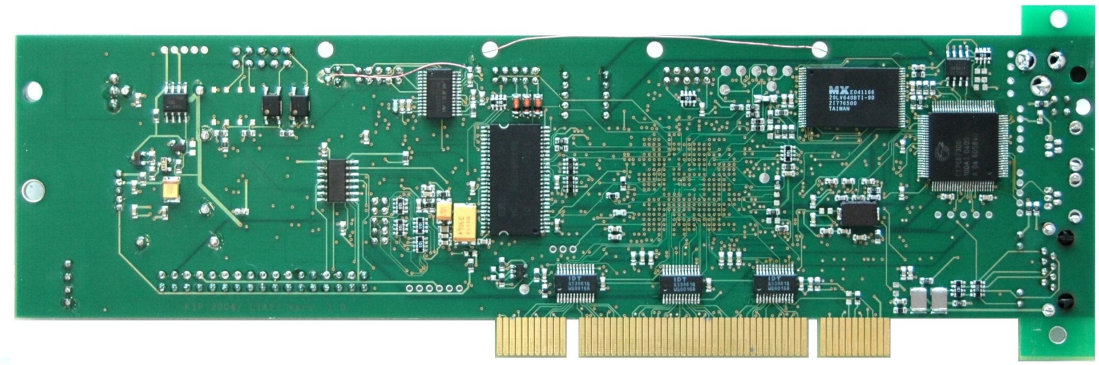


Abbildung 3.2: Photo der USB-Mezzanine-Karte

3.5 CIA2-Karte

Die fertig gestellte CIA2-Karte verfügt über den gleichen Excalibur-Baustein wie die Entwicklerplatine. Der Arbeitsspeicher wurde auf zwei 32 MByte-Module vergrößert, der Flash-Festspeicher auf 16 MByte, ebenfalls in zwei Bausteinen. Außerdem sind mehrere Schnittstellen hinzugekommen. Abbildung 3.3 zeigt eine Photographie der fertig bestückten CIA2-Karte. Am linken Rand sind die Kommunikationsschnittstellen sichtbar: Die Infrarot-Diode, die zwei Mini-USB-Buchsen und die Fast-Ethernet-Schnittstelle. Über dem EPXA1 ist die JTAG-Schnittstelle. Am unteren Rand befindet sich ein Anschluss für ein Diskettenlaufwerk. Dieses kann über ein Flachbandkabel mit dem Controller des Hostrechners verbunden werden. Im Verlauf der Arbeit wurden die Pins auch zum Anschluss des Logikanalysators und des EZ-Host-Debuggers verwendet. Am unteren Rand der Karte sieht man die Pins der Steckleiste für die PCI-Schnittstelle.

(a) Vorderseite der CIA2-Karte



(b) Rückseite der CIA2-Karte

Abbildung 3.3: Photos der CIA2-Karte. (a) Vorderseite, in der Mitte der EPXA1. (b) Rückseite, in der Mitte rechts der EZ-Host.

4 Hardwareanbindung

Die Anbindung des USB-Controllerchips an den Excalibur-Kern war eine der Hauptaufgaben dieser Arbeit. Zunächst galt es, aus der Vielzahl der zur Verfügung stehenden Schnittstellen die geeignetste Kombination auszuwählen. Für den EZ-Host USB Controller fiel die Wahl sehr schnell auf die in Abschnitt 4.2 beschriebene Host-Port-Schnittstelle. Sie erlaubt direkten Zugriff auf den internen Speicher, hat eine maximale Bandbreite von bis zu 16 MByte/s und bietet ein bidirektionales Mailboxregister sowie ein dediziertes Statusregister. Auf der Seite des Excalibur-Bausteins standen effektiv zwei Möglichkeiten zur Verfügung, die Anbindung über die Dual-Ported-SRAM (DPSRAM)-Schnittstelle oder alternativ ein in VHDL[1] implementierter Brückenbaustein (AHB-Slave). Im Verlauf dieser Arbeit wurden beide Varianten implementiert. Sie sollen in den folgenden Abschnitten vorgestellt und dann in Abschnitt 4.5 verglichen werden. Abbildung 4.1 zeigt ein Blockschaltbild der relevanten Komponenten.

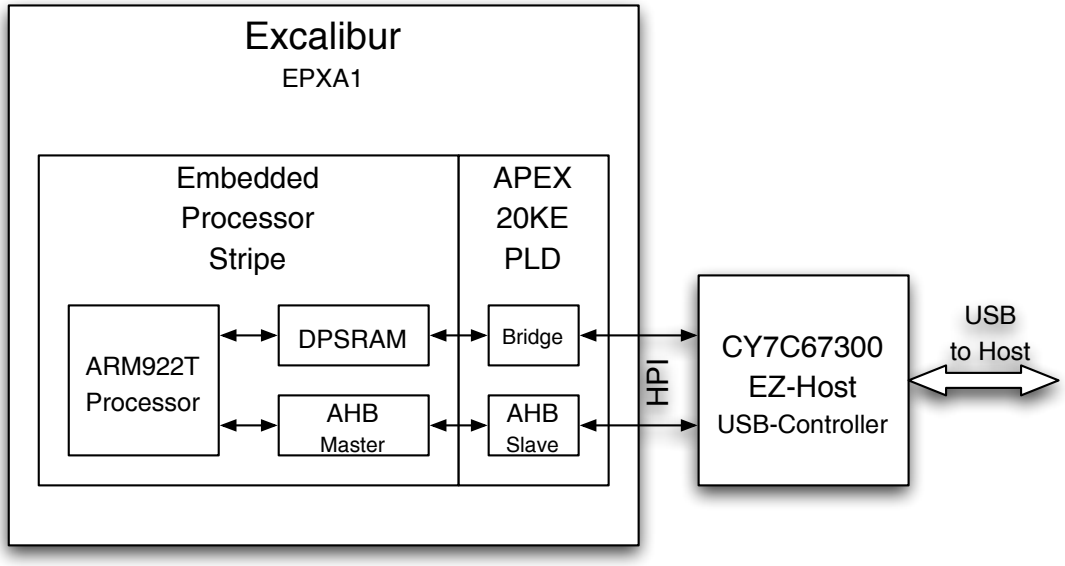


Abbildung 4.1: Blockschaltbild der USB-Anbindung

[1] Die *Very High Speed Integrated Circuit Hardware Description Language* (VHDL) ist eine Hardwarebeschreibungssprache und – zumindest in Europa – De-facto-Standard für den Entwurf und die Simulation von PLD- oder FPGA-basierten Systemen.

4.1 Designflow

Für den Entwurf und die Synthese der entwickelten Hardware wurde die vom Hersteller der Excalibur-Baureihe angebotene Software Quartus II verwendet. Das Werkzeug unterstützt den Entwurf von einfachen, im graphischen Editor aus fertigen Bibliothekskomponenten zusammengesetzten Schaltungen ebenso wie den von komplexen in VHDL beschriebenen Systemen. In dieser Arbeit wurden beide Techniken eingesetzt. Die DPSRAM-Variante wurde mit den graphischen Werkzeugen der Software entworfen. Der AHB-Slave hingegen wurde von Anfang an in VHDL entwickelt und erst zur Synthese in die Software eingelesen. Der Synthesevorgang ist in beiden Fällen identisch, da die Software die graphischen Schaltungsentwürfe zunächst in eine Hardwarebeschreibungssprache umsetzt und anschließend kompiliert. Das Synthesewerkzeug analysiert zunächst die Quellen auf syntaktische Fehler. Werden keine Fehler gefunden, so versucht das Werkzeug die Anweisungen in Hardware umzusetzen. Ist die Synthese erfolgreich, so wird im nächsten Schritt versucht die synthetisierte Hardware auf das verwendete PLD abzubilden. Reichen die Ressourcen des Gerätes nicht aus – sei es, weil zu viele Logikzellen benötigt werden oder weil verwendete Bibliothekskomponenten spezielle, unerfüllbare Anforderungen stellen – so wird der Versuch abgebrochen. Ansonsten generiert das Werkzeug eine Binärdatei, die die gewünschte PLD-Konfiguration beschreibt. Die fertige Binärdatei kann dann mit einem weiteren Werkzeug, dem `exc_flash_programmer`[2] über die JTAG-Schnittstelle in den Flashspeicher der CIA2-Karte geschrieben werden. Während des Startvorgangs des im Excalibur-Baustein integrierten Prozessors wird mit diesen Daten das PLD konfiguriert. Die neue Hardware kann dann getestet und benutzt werden.

4.2 Beschreibung der Host-Port-Schnittstelle

Beim HPI handelt es sich essentiell um einen 16 Bit breiten, bidirektionalen Datenbus (**HPI_IO**) mit sechs zusätzlichen Steuerleitungen:

INT Unterbrechungsleitung. Wird ausgelöst, wenn der interne Prozessor in das Mailboxregister schreibt.

nCS Chip Select (low-aktiv). Signalisiert einen Zugriff auf den Chip.

nRD Read (low-aktiv). Markiert den laufenden Zugriffszyklus als schreibend.

nWR Write (low-aktiv). Markiert den laufenden Zugriffszyklus als lesend.

ADDR Zwei Bit breiter Adressbus. Bestimmt, wie die im laufenden Zyklus auf dem Datenbus anliegenden Daten zu interpretieren sind. Die Kodierung der Bussignale ist in Tabelle 4.1 aufgeführt.

Zugriffe auf das Statusregister oder das Mailboxregisters benötigen nur einen HPI-Zyklus. Das Zielregister wird über den Adressbus bestimmt, im ersten Fall mit dem Buskommando **HPI Status**, im zweiten mit dem Kommando **HPI Mailbox**. Der EZ-Host antwortet bei einem Lesezugriff, indem er den Inhalt des Status- bzw. Mailboxregisters auf

[2] Dieses Kommandozeilenprogramm ist Bestandteil der Quartus II Entwicklungsumgebung.

Ziel	A1	A0	Beschreibung
HPI Data	0	0	Die Daten werden an der beim vorherigen Zyklus bestimmte Adresse in den Speicher geschrieben, bzw. bei einem Lesezugriff von dort gelesen
HPI Mailbox	0	1	Die Daten werden in das dedizierte HPI Mailboxregister geschrieben bzw. von dort gelesen. Bei einem Schreibzugriff wird zusätzlich eine Unterbrechung ausgelöst (HPI Mailbox RX Full)
HPI Address	1	0	Die Daten werden als Zieladresse für den nächsten Lese- oder Schreibzyklus interpretiert und in das Adressregister geschrieben
HPI Status	1	1	Bei einem Lesezugriff wird vom EZ-Host der Inhalt des Statusregisters auf den Bus gelegt. Schreibzugriffe auf das Statusregister sind nicht erlaubt

Tabelle 4.1: Verwendung des HPI-Adressbusses

den Datenbus legt. Bei einem Schreibzugriff auf die Mailbox werden die Daten vom Datenbus gelesen und im Mailboxregister ablegt.[3] Ein Zugriff auf den internen Speicher des EZ-Host erfordert zwei Zyklen. Im ersten Zyklus, der immer schreibend ist, wird auf dem Adressbus das Kommando **HPI Address** gesendet und die Zieladresse auf den Datenbus gelegt. Diese wird in einem EZ-Host-internen Adressregister gespeichert. Im zweiten Zyklus wird der Adressbus auf **HPI Data** gesetzt. Wird danach **nWR** aktiv, so wird der auf dem Datenbus anliegende Wert vom EZ-Host gelesen und in der im Adressregister angegebenen Speicherstelle abgelegt. Wird **nRD** aktiv, so wird der EZ-Host die geforderten Daten aus seinem Speicher lesen und auf den Datenbus legen. Das Adressregister wird in beiden Fällen automatisch inkrementiert. Abbildung 4.2 stellt die Timingdiagramme für Lese- und Schreibzyklen dar. Die Bezeichnungen sind in Tabelle 4.2 erläutert.

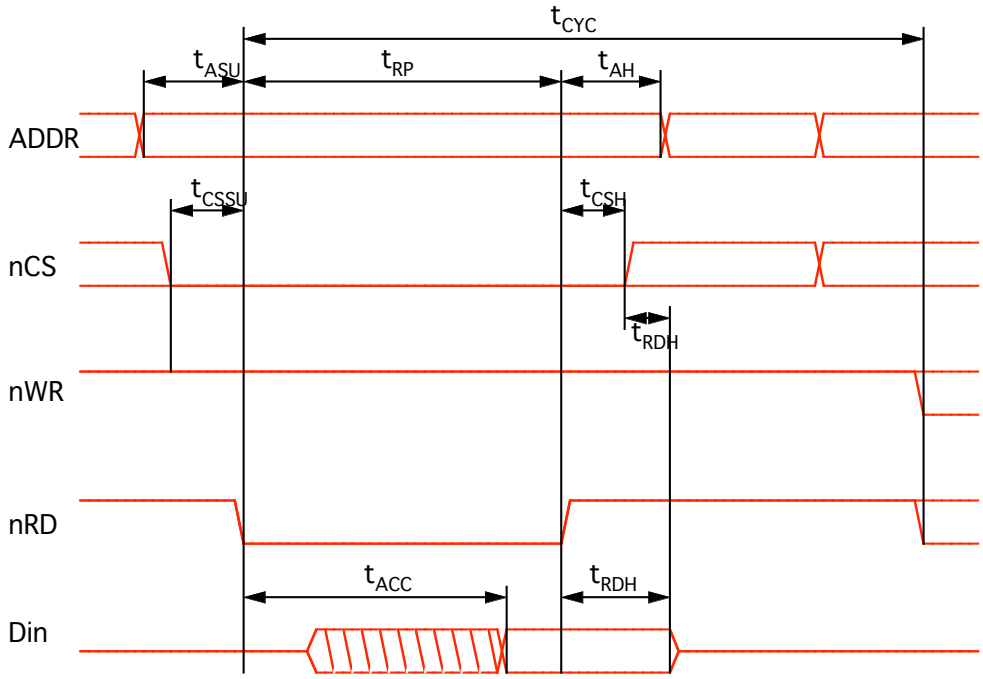
4.3 Beschreibung der DPSRAM-Anbindung

Für einen unkomplizierten und schnellen Zugriff auf extern angebundene Hardware wird von Altera die Dual-ported-SRAM-Schnittstelle vorgeschlagen. In dem Anwendungsbeipiel [AL173] wird eine relativ einfache Anwendung vorgestellt, im Wesentlichen ein Zustandsautomat, der vom Prozessor in den ersten Speicherblock geschriebene Daten ausliest und in den zweiten Speicherblock kopiert. Der in dem angeführten Anwendungsbeispiel beschriebene Entwurf diente als Grundlage für die Entwicklung der Schnittstelle zum EZ-Host. Das Dual-ported SRAM besteht beim verwendeten EPXA1-Modell der Excalibur-Serie aus einem 16 kByte großen integrierten Speicher, der in verschiedenen Konfigurationsmodi betrieben werden kann. Dabei sind Anzahl und Zuordnung der Ports sowie die Datenbreite der Schnittstelle wählbar.[4] Um den Entwurf der USB-Anbindung so einfach wie

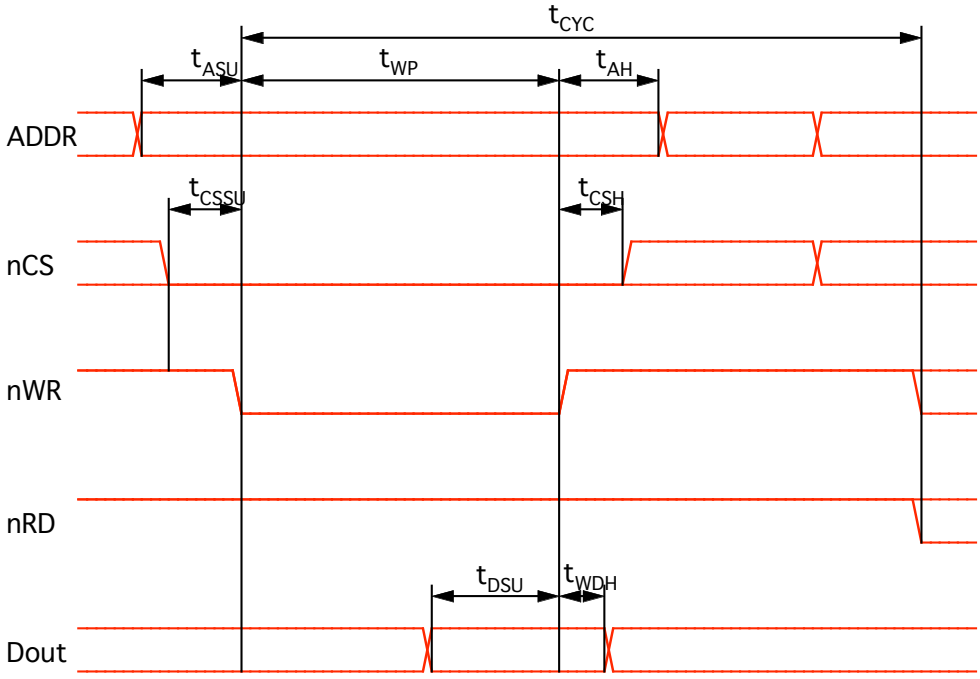
[3] Das Statusregister erlaubt keine Schreibzugriffe von außen.
[4] S. [EDRM, S. 51ff] für eine umfassende Erläuterung der DPSRAM Konfigurationsmöglichkeiten.

Name	Beschreibung
t_{CYC}	Read/Write Cycle Time: Zeit für einen Lese- oder Schreibzyklus, mindestens 6 Takte des EZ-Host-Prozessors
t_{ASU}	Address Set-up: Zeit, die das ADDR-Signal vor aktiv werden des nWR- oder nRD-Signals hat, um stabil zu werden
t_{RP}	Read Pulse Width: Zeit, während der das nRD-Signal aktiv sein muss, mindestens 2 Takte des EZ-Host-Prozessors
t_{AH}	Address Hold: Zeit nach inaktiv werden nWR- oder nRD-Signals, während der das ADDR-Signal weiter gültig sein muss
t_{CSSU}	Chip Select Set-up: Zeit, die das nCS-Signal vor aktiv werden des nWR- oder nRD-Signals hat, um stabil zu werden
t_{CSH}	Chip Select Hold: Zeit nach inaktiv werden nWR- oder nRD-Signals, während der das nCS-Signal weiter gültig sein muss
t_{RDH}	Read Data Hold: Zeit nach inaktiv werden nCS- oder nRD-Signals (es gilt das früher inaktiv werdende Signal), während der die vom EZ-Host gelieferten Daten gültig sind. Höchstens 7 ns
t_{ACC}	Data Access Time: Zeit nach aktiv werden des nRD-Signals, die der EZ-Host benötigt, um auf die Daten zuzugreifen. Höchstens einen Takt des EZ-Host-Prozessors
t_{DSU}	Data Set-up: Zeit, während der die zu schreibenden Daten vor inaktiv werden des nWR-Signals stabil sein müssen. Mindestens 6 ns
t_{WP}	Write Pulse Width: Zeit, während der das nWR-Signal aktiv sein muss, mindestens 2 Takte des EZ-Host-Prozessors
t_{WDH}	Write Data Hold: Zeit nach inaktiv werden nWR-Signals, während der die zu schreibenden Daten weiter gültig sein müssen. Mindestens 2 ns

Tabelle 4.2: Beschreibung der HPI-Timing-Konstanten



(a) HPI-Lesezyklus



(b) HPI-Schreibzyklus

Abbildung 4.2: HPI-Timingdiagramme für Lesezyklen (a) und Schreibzyklen (b), Quelle: [CEDS]

möglich zu gestalten, wurde als Konfiguration ein 4 Kilowörter fassender Speicher in 32-Bit-Anordnung gewählt. Hierbei werden die unteren 16 Bit als bidirektionaler Datenbus verwendet, die oberen 16 Bit für die im HPI-Protokoll vorgesehenen Steuerleitungen. Die einfachste Implementierung der Anbindung ist, die kombinierten Daten- und Steuersignale über den prozessorseitigen Speicherport in die unterste Speicherstelle zu schreiben. Über den PLD-Port werden sie dann an den EZ-Host weitergeleitet. Außerdem wird mindestens ein zusätzliches externes Signal benötigt, um die Datenflussrichtung zu steuern. In dieser einfachen Form würde allerdings der restliche freie DPSRAM-Speicher nicht genutzt, sondern nur die untersten 32 Bit. Die implementierte Lösung verwendet den gesamten Speicher und besteht aus einem im PLD realisierten Zähler (LPM_COUNTER) und einem hochohmigen Bus samt zugehöriger Brücke (LPM_BUSTRI).[5] Die vier im EPXA1 zur Verfügung stehenden GPIOs werden als externe Steuerleitungen verwendet. Die Funktion der einzelnen GPIOs ist in Tabelle 4.3 beschrieben. Abbildung 4.3 zeigt ein Schaltbild des

GPIO	**Verwendung wenn aktiv**
0	Stellt den Bus auf „Lesen“, d.h. die vom EZ-Host eingehenden Daten werden an den Eingang des DPSRAM angelegt
1	Aktiviert das Write Enable Signal des PLD-seitigen Speicherports, d.h. die am Eingang anliegenden Daten werden an der durch den Zähler definierten Adresse in den Speicher geschrieben
2	Inkrementiert den Adresszähler
3	Setzt den Adresszähler auf 0 zurück

Tabelle 4.3: Verwendung der Excalibur-GPIOs im DPSRAM-Entwurf

Entwurfs. Die Signal- und Komponentenbezeichnungen in den folgenden Erläuterungen beziehen sich auf dieses Schaltbild. Da die DPSRAM-Schnittstelle über getrennte Ein- und Ausgänge verfügt, wird die hochohmige Brücke gpio_tri benötigt, um den bidirektionalen HPI-Bus, CY_GPIO_BIDIR, in die zwei unidirektionalen DPSRAM-Datenbusse dpram_in und dpram_out zu überführen. Lese- und Schreibzugriffe schließen sich gegenseitig aus. Die Datenflussrichtung kann also mit einem einzelnen Signal und einem Inverter gesteuert werden. Ist gp_out[0] inaktiv, wird der Ausgang des Speichers auf den bidirektionalen Datenbus gelegt, der Eingang wird hochohmig geschaltet. Ist gp_out[0] aktiv, werden die vom EZ-Host auf den Bus gelegten Daten mit dem Eingang des Speichers verbunden, der Ausgang wird ignoriert. Prinzipiell könnte man dieses Signal auch verwenden, um den Write-Enable-Eingang der Speicherschnittstelle, dp0_portawe, zu schalten. Dies führt in der Praxis aber zu Laufzeitproblemen, daher wird mit gp_out[1] ein eigenes Signal dafür verwendet.

Bei der DPSRAM-Schnittstelle ist das HPI-Protokoll ist vollständig in Software implementiert. Im Code der in Abschnitt 5.1 beschriebenen Funktionen werden die Zustände in sequentieller Folge in den Speicher geschrieben und damit die zugehörigen HPI-Signale aktiviert. Wo nötig, wird zusätzlich das GPIO-Register beschrieben. Diese (finale) Implementierung der DPSRAM-Anbindung nutzt die gesamten vorhandenen 4 Kilowörter des

[5] Alle Komponenten stammen aus der von Altera mitgelieferten Bibliothek, die Namen in Klammern bezeichnen die Altera-eigene Nomenklatur.

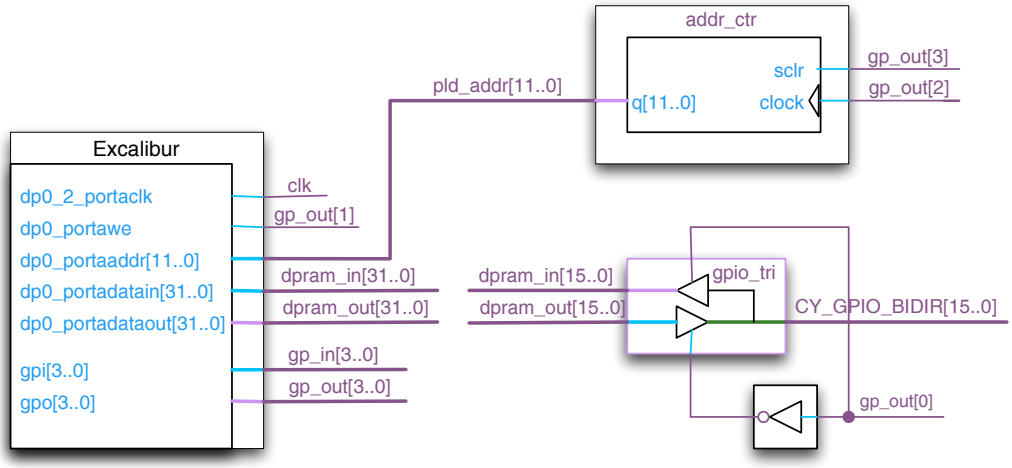


Abbildung 4.3: Blockschaltbild der DPSRAM-Anbindung

DPSRAM-Speichers, da nicht nur die unterste Adresse verwendet wird, sondern über den Adresszähler addr_ctr jede Speicheradresse erreicht werden kann. Der Ausgang des Zählers ist mit dem Adresseingang dp0_portaaddr[11..0] des Speichers verbunden. Durch aktivieren von gp_out[2] wird der Zähler und damit die Zieladresse erhöht. Der Adresszähler kann über gp_out[3] zurückgesetzt werden.

Zwei Probleme ergeben sich durch den Datenzwischenspeicher (Cache) des prozessorseitigen Speicherports:

1. Die Steuersignale, die auf den oberen 16 Bit des Datenwortes liegen, werden nicht synchron an den Speicher (und damit an die PLD-Hardware an Port 2) weitergereicht, sondern im Cache gespeichert.

2. Bei Lesezugriffen werden die Antwortdaten vom PLD direkt in den Speicher geschrieben, vom Prozessor aber aus dem Cache gelesen.

Der Cache auf der Prozessorseite verfügt nicht über die Information, ob eine externe Hardware auf der PLD-Seite angeschlossen ist. Wird der Speicher PLD-seitig beschrieben, sind Speicher und Cache inkohärent. Hier führte eine Anfrage an die Firma Altera zu Abhilfe: durch Modifikation[6] einiger Codezeilen in der Systeminitialisierungsdatei `crt0.s` wird der Cache deaktiviert. Laut Kommentaren im Quelltext wird durch diese Modifikation allerdings auch die Speicherverwaltungseinheit (MMU) deaktiviert – dies wäre eine nicht tolerierbare Einschränkung der Systemeigenschaften. Die Spezifikation der Prozessorinstruktionen belegt, dass Cache und MMU durch verschiedene Bits in einem Register gesteuert

[6] Genauer: durch Auskommentieren der folgenden Codezeilen:
```
MRC p15,0,r0,c1,c0,0
ORR r0,r0,#5 /* enable DCache and MMU */
MCR p15,0,r0,c1,c0,0
```
s. [AARM] für Erläuterungen der Instruktionen.

werden, sie ließen sich also auch getrennt (de)aktivieren. Dies wurde jedoch nicht getestet, da die von Altera vorgeschlagene Vorgehensweise das Problem löste. Das gleichzeitige Deaktivieren von MMU und Cache wurde in Kauf genommen, da die Nachteile für diese Phase der Arbeit, das Testen der physikalischen Anbindung des EZ-Host-Chips, nicht von Bedeutung war.

4.4 Anbindung über den AHB-Slave

Die für die CIA-Karte verwendete Entwicklungsumgebung macht die Einbindung von Komponenten, die auf der AHB-Spezifikation beruhen, sehr einfach.

4.4.1 Überblick über die AHB-Spezifikationen

Der *Advanced High-Performance Bus* (AHB) wurde im Jahre 1999 von der Firma ARM als Teil der AMBA-Spezifikationen[7] vorgestellt. Als Anwendungsgebiet ist die Anbindung externer Koprozessoren oder Speicherschnittstellen vorgesehen. Der AHB ist für hohen Datendurchsatz konzipiert. Daher verfügt er über getrennte Lese-, Schreib- und Adressbusse und unterstützt den beschleunigten Transfer großer, konsekutiv im Speicher vorliegender Datenmengen (Bursts). Tabelle 4.4 gibt einen Überblick über die Signale des AHB.

Ein normaler Lese- oder Schreibzyklus gliedert sich in eine Adress- und eine Datenphase. Da separate Adress- und Datenbusse vorliegen, werden bei Bursts die Adresse des folgenden Zyklus synchron mit den Daten des aktuellen Zyklus gesendet. Der Slave kann über das HREADY-Signal seine Bereitschaft, Daten zu empfangen, signalisieren. Empfängt der Master ein inaktives HREADY, wird er die Datenphase verlängern.[9] Während der Adressphase werden parallel zur Adresse auch die relevanten Steuersignale HWRITE, HSEL und HREADY übertragen. Da erst zu diesem Zeitpunkt klar wird, welcher der angeschlossenen Slaves das Ziel wird, müssen alle potentiellen Empfänger diese Signale in ein Register speichern. Bei der nächsten steigenden Flanke von HCLK liegen bereits die Daten auf dem Bus und müssen, je nach Zustand von HWRITE, vom Slave oder vom Master gelesen werden. Der Slave signalisiert dann über HRESP, ob der Datentransfer erfolgreich war oder abgebrochen wurde.

Wird auf einem 32 Bit breiten Bus ein 16-Bit-Datenzyklus gestartet, so entscheidet die Adresse darüber, ob die Nutzdaten auf den oberen oder unteren 16 Bit des Datenbusses geschrieben bzw. gelesen werden. Tabellen 3-6 und 3-7 in [AMBA] geben Aufschluss über die – je nach Endianness unterschiedlichen – verwendeten Leitungen.

4.4.2 Beschreibung des AHB-Slaves

An die zu implementierende Hardware wurden mehrere Anforderungen gestellt:

[7] Advanced Microprocessor Bus Architecture, s. [AMBA].

[8] Die Altera-Implementation des AHB weicht an dieser Stelle von den Spezifikationen ab: das HSIZE-Signal ist nur zwei Bit breit. Die Spezifikation sieht Datenblöcke von bis zu 1024 Byte bei einem drei Bit breiten HSIZE-Signal vor.

[9] Bei Bursts kann auf diese Weise auch die Adressphase der Folgezugriffe verlängert werden.

[10] In der vorliegenden Implementierung des AHB-Slaves werden RETRY- und SPLIT-Transfers nicht unterstützt.

Name	Quelle	Beschreibung
HCLK	–	Zentrales Taktsignal, Zeitangaben sind relativ zur steigenden Flanke
HRESETn	–	Invertiertes Bus Reset Signal
HADDR[31:0]	Master	32 Bit breiter Adressbus
HTRANS[1:0]	Master	Bezeichnet die Art des aktuellen Transfers, s. Tabelle 4.7
HWRITE	Master	Bezeichnet die Transferrichtung. Wenn aktiv, erfolgt aus Sicht des Masters ein Schreibzugriff, ein Lesezugriff wenn inaktiv
HSIZE[2:0][8]	Master	Zeigt die Datenbreite des Transfers an, s. Tabelle 4.6
HBURST[2:0]	Master	Signalisiert einen Burst bzw. Art und Länge des Burst-Transfers
HWDATA[31:0]	Master	32 Bit breiter Datenbus für Schreibzugriffe
HRDATA[31:0]	Slave	32 Bit breiter Datenbus für Lesezugriffe
HSELx	Dekodierer	Chip Select Signal. Der im Excalibur realisierte AHB-Dekodierer verwendet die obersten Adressbits, um ein individuelles HSEL-Signal für jeden Slave zu generieren
HREADY(I)	Slave	Mit HREADY signalisiert der Slave, ob er zum Datentransfer bereit ist. Ein inaktives Signal veranlasst den Master dazu, die Datenphase zu verlängern
HREADY(O)	Master	Auch der Master verfügt über ein HREADY-Signal, hiermit wird dem Slave signalisiert, dass er sich auf einen Transfer vorbereiten soll
HRESP[1:0]	Slave	Beschreibt den Status des aktuellen Transfers aus Sicht des Slaves, s. Tabelle 4.5

Tabelle 4.4: Überblick über die AHB-Signale, Quelle:[AMBA]

1. Der Entwurf muss sowohl die AHB- als auch die HPI-Spezifikationen einhalten.

2. Die in der AHB-Schnittstelle parallelen Adress-, Lese- und Schreibbusse müssen in den einzelnen, gemultiplexten HPI-Bus überführt werden.

3. Der Entwurf sollte möglichst allgemein gehalten sein, um den Einsatz auf anderen AHB-basierten Systemen zu ermöglichen.

4. Die Datenrate sollte dabei so hoch wie möglich sein.

Zur Umsetzung dieser Anforderungen wurde ein Zustandsautomat entwickelt. In der Adressphase wird die ankommende Adresse gespeichert, ebenso die Steuersignale HWRITE und HSEL. Danach geht der Automat in die Datenphase. Hier werden die Daten gespeichert und danach Adresse und Daten sequentiell über den HPI-Bus an den EZ-Host geleitet. Protokollfehler werden mit einer den Spezifikationen entsprechenden Antwort erwidert (HRESP = ERROR), bei einem Lesezugriffsfehler wird zur Erkennung die Datenfolge 0xDEAD-

HRESP[1]	HRESP[0]	Beschreibung
0	0	**OKAY** Wird verwendet, um einen erfolgreichen Transfer anzuzeigen (HREADY aktiv) oder eine reguläre Verlängerung der Datenphase zu signalisieren (HREADY inaktiv)
0	1	**ERROR** Signalisiert einen Fehler bei der Übertragung. Muss über 2 Takte gehalten werden
1	0	**RETRY**[10]Zeigt dem Master an, dass ein Transfer noch nicht beendet werden konnte. Der Master wird den Transferversuch wiederholen, bis er erfolgreich war. Muss über 2 Takte gehalten werden
1	1	**SPLIT**[10] Signalisiert einen unbeendeten Transfer an. Der Master wird den Bus freigeben und den Transfer wiederholen, wenn er erneut Zugriff auf den Bus erhält. Muss über 2 Takte gehalten werden

Tabelle 4.5: Kodierung des AHB-Signals HRESP

HSIZE[1]	HSIZE[0]	Beschreibung
0	0	8 Bit (Byte) HSIZE_BYTE
0	1	16 Bit (Halfword)HSIZE_HWORD
1	0	32 Bit (Word) HSIZE_WORD
1	1	64 Bit (Doubleword) HSIZE_DWORD

Tabelle 4.6: Kodierung des AHB-Signals HSIZE

FACE auf den HRDATA-Bus gelegt. Aus praktischen Gründen werden zwei Zustandsautomaten parallel betrieben: der erste implementiert das AHB-Protokoll, der zweite das HPI-Protokoll. Dies vereinfacht die Entwicklungsarbeit, da die Protokollkonformität getrennt überprüft werden kann.

4.4.3 Beschreibung des Zustandsautomaten

Nach Durchlaufen des obligatorischen Resetzustandes befindet sich der AHB-Automat in der Adressphase (ahb_state = ADDRESS_PHASE), der HPI-Automat im Initialisierungszustand (bridge_state = XFER_ADDRESS_INIT). Solange der AHB-Automat in diesem Zustand bleibt, werden Adresse und Steuersignale kontinuierlich in ein Register geschrieben. Der AHB-Automat geht genau dann in die Datenphase, wenn

a) das Chip-Select Signal aktiv ist (HSEL = '1'),

b) die Transaktionsart nicht-sequentiell ist (HTRANS = HTRANS_NONSEQ) und

c) die Datenbreite 16 Bit beträgt (HSIZE = HSIZE_HWORD).

Sind alle diese Konditionen erfüllt, so wird der HPI-Automat aktiviert und durchläuft seine Zustände, beginnend bei bridge_state = XFER_ADDRESS_INIT. In jedem Zustand

HTRANS[1]	HTRANS[0]	Beschreibung
0	0	**IDLE** Wurde einem AHB-Master der Zugriff auf den Bus gewährt, und hat der Master keine Daten, so markiert er den Transfer als IDLE. Wird nur ein Master verwendet, so hat er immer Buszugriff. In diesem Fall werden IDLE-Transfers der Normalfall sein
0	1	**BUSY** Prinzipiell erlaubt die AHB-Spezifikation sequentielle Zugriffe – so genannte Burst-Transfers. Kann der aktive AHB-Master in einem Burst-Transfer vorübergehend keine Daten senden, so signalisiert er BUSY. Der betroffene AHB-Slave sollte dann den Transfer ignorieren und im nächsten Takt mit OKAY antworten. Der vorliegende AHB-Slave unterstützt keine Burst-Transfers, daher wird an dieser Stelle nicht weiter auf diese Funktionalität eingegangen. Für umfangreiche Information sei auf [AMBA, Abschnitt 3.6] verwiesen
1	0	**NONSEQ** Dies ist der Normfall für einen Datentransfer. Genau genommen sehen die Spezifikationen ein einzelnes Datenpaket als einen Burst mit Länge 1
1	1	**SEQ** In diesem Fall wird das erste Datenpaket als NONSEQ markiert, alle folgenden als SEQ

Tabelle 4.7: Kodierung des AHB-Signals HTRANS

werden die für das Protokoll notwendigen HPI-Signale (de)aktiviert. Im letzten Zustand des HPI-Automaten (bridge_state = XFER_FINISH) wird das HREADY-Signal aktiviert. Dies signalisiert dem AHB-Automaten das Ende der Datenphase[11]. Er geht wieder in die Adressphase über und versetzt dadurch auch den HPI-Automaten wieder in den Anfangszustand (XFER_ADDRESS_INIT). Tabelle 4.8 erläutert die einzelnen Zustände des HPI-Automaten. Wird ein Burst-Transfer angefordert (HTRANS = HTRANS_SEQ) oder ist die Datenbreite nicht HSIZE_HWORD[12], so geht der AHB-Automat in den Fehlerzustand über. Dort wird die Fehlerantwort, wie von den Spezifikationen gefordert, über zwei Takte gehalten. Danach kehrt der Automat in die Adressphase zurück. Den zugehörigen Zustandsgraphen zeigt Abbildung 4.4.

4.4.4 Simulation

Der fertiggestellte AHB-Slave wurde mit Hilfe des von Altera bereitgestellten AHB-Simulationsmodells in der softwarebasierten Simulationsumgebung ModelSim simuliert.[13] Die Ergebnisse der Simulation sind in Unterabschnitt B.1.1 dargestellt. Das Modell ist natürlich

[11] Die zweite Kondition für den Übergang Adressphase→Datenphase, HRESP = HRESP_OKAY ist immer erfüllt.

[12] 16 Bit, s. Tabelle 4.6.

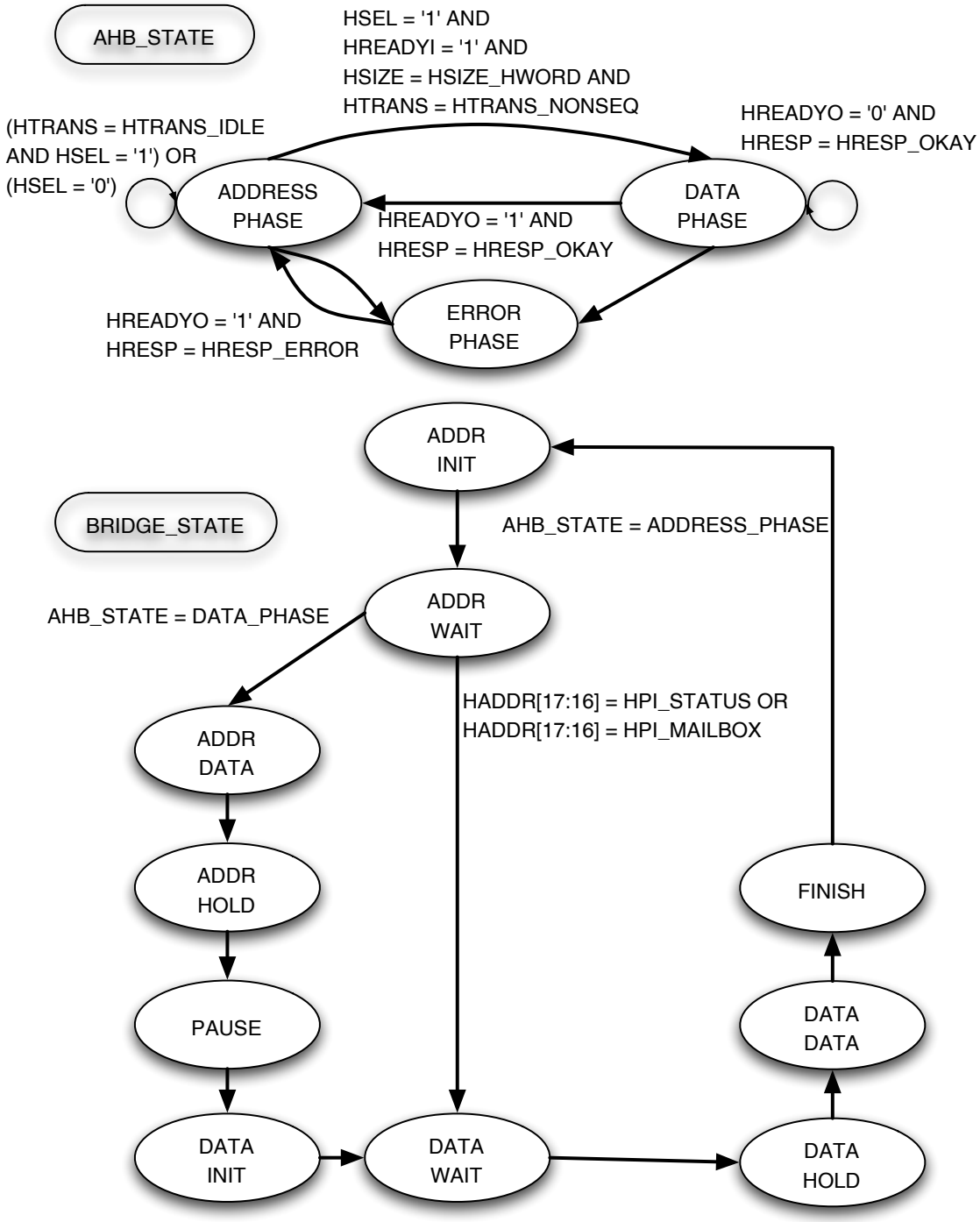


Abbildung 4.4: Übergangsdiagramm der Zustandsautomaten im AHB-Slave

Zustand	Beschreibung
XFER_ADDR_INIT	Initialzustand. Dekodiert die oberen Adressbits und setzt das HPI_ADDR-Signal (s. Tabelle 4.1) sowie den Folgezustand entsprechend. Setzt das Datenregister data_reg, je nach Adressoffset auf die oberen oder unteren 16 Bit von HWDATA
XFER_ADDR_WAIT	Aktiviert das HPI_WR_N-Signal für den Address-Schreibzyklus
XFER_ADDR_DATA	Legt die im Adressregister addr_reg gespeicherte Adresse auf den bidirektionalen Bus HPI_IO
XFER_ADDR_HOLD	Deaktiviert das HPI_WR_N-Signal für den Address-Schreibzyklus
XFER_PAUSE	In diesem Zustand wird an den Ausgängen nichts verändert, er dient nur zur Einhaltung der Timing-Anforderungen
XFER_DATA_INIT	Setzt das HPI_ADDR-Signal auf HPI_DATA. Der Datenbus HPI_IO wird hochohmig geschaltet
XFER_DATA_WAIT	Für den Datenzyklus wird HPI_RD_N auf das gespeicherte HWRITE-Signal write_reg geschaltet, HPI_WR_N auf den invertierten Wert
XFER_DATA_DATA	Legt die im Datenregister data_reg gespeicherten Daten auf den bidirektionalen Bus HPI_IO, wenn es sich um einen Schreibzugriff handelt (write_reg = '1'). Ansonsten wird das Datenregister auf 0x0000 gesetzt und HPI_IO hochohmig gehalten
XFER_DATA_HOLD	Deaktiviert das HPI_WR_N-Signal für einen Schreibzyklus oder das HPI_RD_N-Signal für einen Lesezyklus. Im letzteren Fall werden zusätzlich die am bidirektionalen Bus anliegenden Daten auf die oberen und unteren 16 Bit des HRDATA-Busses gelegt – s.a. [AMBA, Abschnitt 3.15]
XFER_FINISH	Beendet die AHB-Datenphase, indem das HREADYO-Signal aktiviert wird. Dies veranlasst den AHB-Automaten, in die Adressphase zurückzukehren

Tabelle 4.8: Zustände des HPI-Zustandsautomaten

nicht in der Lage, die Reaktion des EZ-Host zu simulieren, sondern dient nur zur Verifikation des korrekten Ablaufs des Zustandsautomaten sowie der Timing-Anforderungen, wie sie in Abbildung 4.2 dargestellt sind.

4.4.5 Leistungsdaten des AHB-Slaves

Cypress gibt in [CEDS] den theoretischen Maximaldurchsatz der HPI-Schnittstelle mit 16 MByte/s an. Da jeder vollständige Speicherzugriff jedoch je einen Adress- und einen Datenzyklus benötigt, ist der eigentliche maximale Nutzdatendurchsatz um den Faktor 2 kleiner, liegt also bei 8 MByte/s.[14] Die konkrete Messung dieses Wertes stößt auf zwei Schwierigkeiten:

1. Eine Messung des Durchsatzes kann nur mit Mitteln des Betriebssystems stattfinden. Der Messwert beinhaltet also den kombinierten Overhead von Betriebssystem, Kommunikationssoftware und Bibliotheksaufrufen.

2. Der freie Speicher des EZ-Host ist auf ca. 16 kByte begrenzt.[15] Bei maximalem Durchsatz würde ein Schreibzugriff auf den gesamten freien Speicher demnach ca. eine Millisekunde benötigen. Dies liegt unter der Messgenauigkeit des verwendeten `gettimeofday`-Aufrufs. Daher werden pro Messung mehrere Leseaufrufe durchgeführt. Die Messdauer wird dabei möglichst konstant gehalten.

Zum Testen wurde eine weitere ECCL-Anwendung geschrieben. Diese durchläuft eine Schleife, in der Datenblöcke an den EZ-Host geschrieben werden. Die Blockgröße wird zwischen den Schleifendurchläufen jeweils verdoppelt. In einer inneren Schleife werden pro Blockgröße mehrere Schreibaufrufe durchgeführt. Die Messdauer wird dabei möglichst konstant gehalten, indem die Anzahl der Wiederholungen bei der Verdopplung der Blockgröße jeweils halbiert wird. Das Programm wurde anschließend modifiziert, um den Durchsatz bei Lesezugriffen zu messen. Wie aus Abbildung 4.5 deutlich wird, steigt der Durchsatz mit der Blockgröße schnell an und erreicht dann einen Maximalwert. Der erreichte Durchsatz von ca. 1 MByte/s bei Lese- und 2 MByte/s bei Schreibzugriffen liegt unter dem theoretischen Maximum der Spezifikation, ist allerdings für praktische Zwecke vollkommen ausreichend, denn

1. verwendet die vorgesehene Hauptanwendung – USB-basierter Massenspeicher – große Blockgrößen, arbeitet also im Plateaubereich der Hardware,

2. bildet die verfügbare Speichergröße des EZ-Host eine deutlich stärkere Begrenzung der Leistung und

3. werden Schreibzugriffe den überwiegenden Anteil der Datentransfers bilden, da der implementierte Massenspeicher nur Lesezugriffe zulassen soll. Der AHB-Slave wird

[13] Da der Hardwareentwurf für die DPSRAM-Anbindung aus Bibliothekskomponenten besteht und die Hauptlast des HPI-Protokolls in Software abgearbeitet wird, wurde auf eine Simulation verzichtet.

[14] Lese- und Schreibzyklen benötigen jeweils 6 Takte. Die von Cypress angegebene Formel zur Berechnung des Durchsatzes lautet: 48 MHz / 6 = 8 MHz ⇒ 16,0 MByte/s bei Daten in 16-Bit-Anordnung.

[15] Der für Anwendungscode und -daten reservierte Bereich des internen Speichers liegt im Segment 0x3FFF–0x04A4, ist also 15196 Bytes groß.

also hauptsächlich Daten zur Weiterleitung an den Host in den Speicher des EZ-Host schreiben.

Der verwendete EZ-Host-Chip ist USB 2.0 kompatibel, unterstützt allerdings nur die seit USB 1.1 verfügbaren Low- und Full-Speed-Modi. Der maximale Durchsatz liegt für Full-Speed bei 12 MBit/s. Dieser wird von dem AHB-Slave bei Blockgrößen von 8 Byte erreicht. Eine weitere Steigerung der Durchsatzraten wäre also für diese Anwendung nicht notwendig. Da die Hardware jedoch auch für weitere USB-Anwendungen nutzbar sein soll, könnten sich für stark abweichende Anforderungen[16] an den Durchsatz Engpässe ergeben. Dann müssten eventuell weitere Optimierungsanstrengungen unternommen werden. Der mit Quartus II synthetisierte Entwurf des AHB-Slaves belegt laut Report des Synthesewerkzeugs 164 Logikzellen und 92 Register, das entspricht weniger als 4% der im EPXA1 verfügbaren Ressourcen.

4.5 Diskussion der Vor- und Nachteile

Schon zu Beginn der Arbeit an der Hardwareschnittstelle war klar, dass ein AHB-Slave auf lange Sicht die geeignetere Lösung ist. Dies hat mehrere Gründe:

1. Die Verwendung der DPSRAM-Schnittstelle bindet Speicherressourcen, die anderweitig sinnvoller zu nutzen wären.

2. Beim DPSRAM muss das HPI-Protokoll in Software realisiert werden, dies belastet den Prozessorkern des Excalibur-Chips zusätzlich.

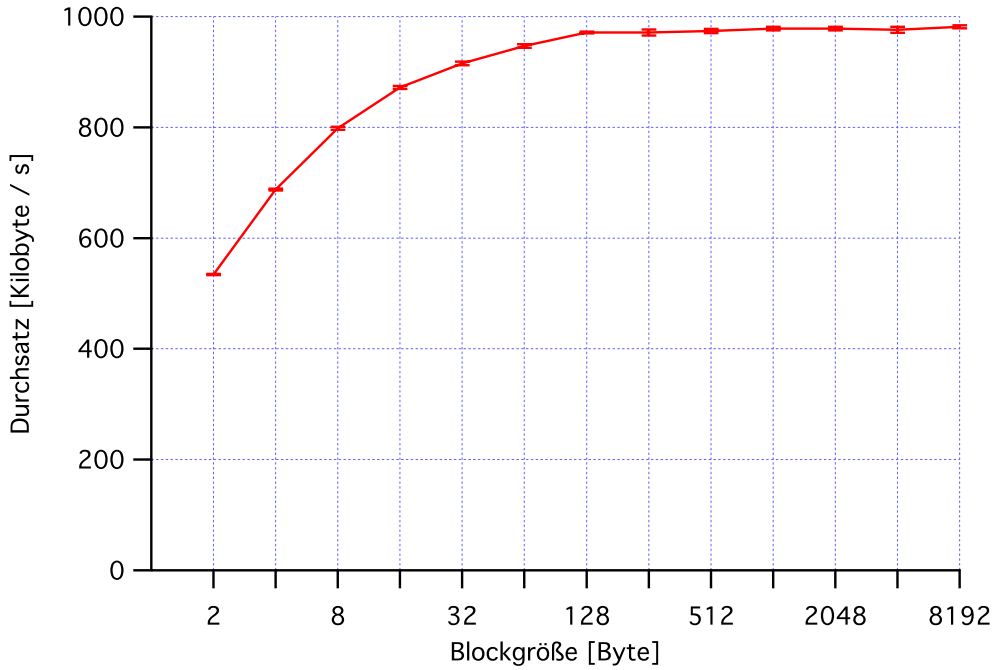
3. Das AHB-Protokoll wird von vielen Herstellern unterstützt. Der AHB-Slave kann leichter an die jeweiligen Rahmenbedingungen angepasst und so in anderen Projekten weiter verwendet werden.

4. Der AHB-Slave bindet den EZ-Host einfach in Form eines weiteren Speicherbereichs ein, der Übergang vom AHB- ins HPI-Protokoll wird vollständig im PLD abgehandelt und entlastet damit den ARM-Prozessor.

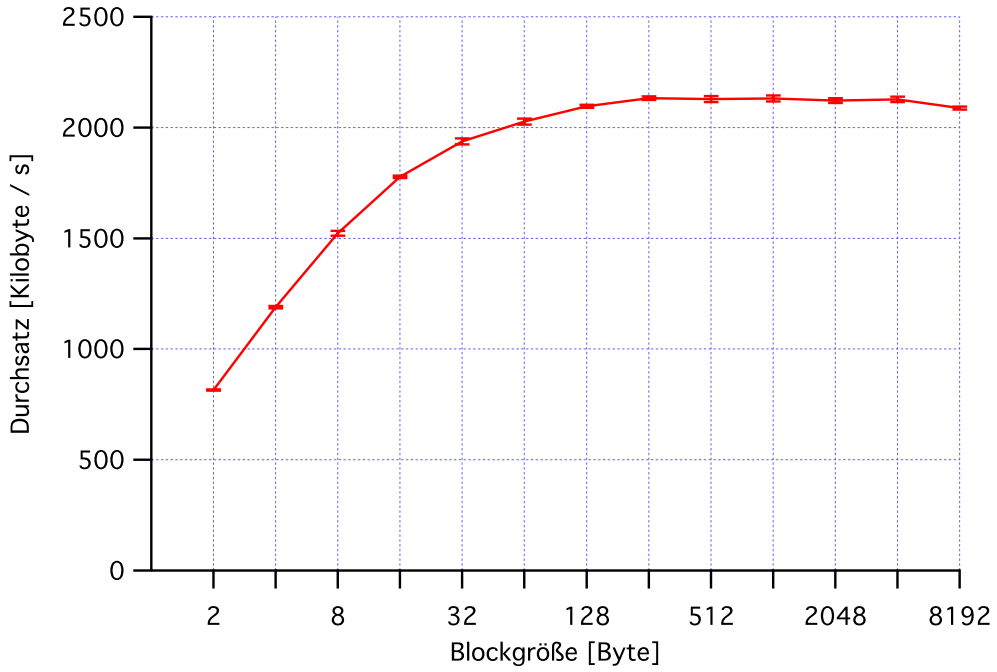
Trotzdem war die Entwicklung der DPSRAM-Schnittstelle sinnvoll, da sie schneller verfügbar war und damit eine Möglichkeit bot, die zugrunde liegende Hardware zu testen. Dies war besonders während des Entwicklungsprozesses der CIA2-Karte hilfreich, da letzte Korrekturen am Layout (den USB-Chip betreffend) frühzeitig entschieden werden konnten. Zudem steht für den Fall knapper PLD-Ressourcen schnell eine alternative Hardware-Anbindung zur Verfügung. Folgerichtig wurde die Kommunikationssoftware auch nach Fertigstellung des AHB-Slaves nicht vollständig umgeschrieben, sondern um die benötigten neuen Funktionen ergänzt.[17]

[16] Eine USB-Tastatur etwa würde deutlich kleinere Blockgrößen bei häufigeren Transfers erfordern.

[17] Allerdings lag die Priorität nun bei den neuen Funktionen. Es besteht also die Möglichkeit, dass die notwendigen Modifikationen Inkompatibilitäten in der Software herbeigeführt haben.



(a) Durchsatz bei Lesezugriffen



(b) Durchsatz bei Schreibzugriffen

Abbildung 4.5: Datendurchsatz des AHB-Slaves in Abhängigkeit von der Blockgröße für Lesezugriffe (a) und Schreibzugriffe (b)

5 Softwareanbindung

Bereits während der Entwicklungsphase der Hardwareanbindung ergab sich die Notwendigkeit einer geeigneten Software-Schnittstelle. Diese sollte unter dem Betriebssystem der CIA2-Karte, einer an den ARM-Kern des EPXA1 angepassten Linuxversion aus der 2.4er Kernelserie, funktionsfähig und in Form einer Bibliothek von Funktionsaufrufen von beliebigen Applikationen wiederverwendbar sein. Zur Unterstützung bei der Entwicklung der Firmware (s. Kapitel 6) musste zudem ein Werkzeug geschrieben werden um EZ-Host-Register auszulesen oder Binärdateien in den Speicher des EZ-Host zu laden. In Abschnitt 5.1 wird die Bibliothek in ihren Funktionen vorgestellt. Danach werden in Unterabschnitt 5.1.2 die bei der Kompilierung der Bibliothek nutzbaren Compilerdirektiven erläutert. Im folgenden Unterabschnitt 5.1.3 wird erläutert, welche Details bei der Nutzung der Bibliothek zu beachten sind. Abschnitt 5.2 beschreibt als Anwendungsbeispiel das auf der Bibliothek basierende Monitorwerkzeug CYMON. Der letzte Abschnitt behandelt Funktionalitäts- und Leistungstest der Bibliothek und des Monitorwerkzeugs.

5.1 Excalibur Cypress Communication Library (ECCL)

Die im Rahmen dieser Arbeit entwickelte Linux-Bibliothek ECCL besteht aus einem Satz von sieben Funktionen, welche die über HPI zur Verfügung stehenden Funktionalität des EZ-Host nutzbar machen. Darüber hinaus beinhaltet die ECCL drei Funktionen die benötigt werden, um die unter Linux zugriffsbeschränkte Hardware ansprechen zu können.[1]

5.1.1 Funktionen

getMappedHPIBase Gibt einen Zeiger auf den von `hpiInit` gespiegelten Speicherbereich für den AHB-Slave zurück.

getMappedRegisterBase Gibt einen Zeiger auf den von `hpiInit` gespiegelten Speicherbereich für die Excalibur-Register zurück.

hpiInit Initialisiert die Bibliothek. Spiegelt den Adressbereich des AHB-Slaves sowie der Excalibur-Register in zwei vom Programmierer frei beschreibbare Speicherbereiche.

hpiRead Liest Daten aus dem Speicher des EZ-Host-Chips. Die Startadresse und Anzahl der zu lesenden Datenwörter ist variabel.

hpiReadMailbox Liest das EZ-Host Mailboxregister.

hpiReset Setzt den AHB-Slave und den EZ-Host in den Initialzustand zurück.

[1] Diese Problematik wird in Unterabschnitt 5.1.3 besprochen.

hpiStatus Liest das EZ-Host Statusregister.

hpiWrite Schreibt Daten in den Speicher des EZ-Host. Die Startadresse und Anzahl der zu schreibenden Datenwörter ist variabel. Ein Zeiger auf einen Speicherbereich mit zu schreibenden Daten muss übergeben werden.

hpiWriteMailbox Schreibt ein 16-Bit-Datenwort in das EZ-Host Mailboxregister.

unmapRegions Gibt die von `hpiInit` gespiegelten Speicherbereiche wieder frei. Diese Funktion sollte stets der letzte Bibliotheksaufruf sein.

Eine vollständige Beschreibung der Funktionen findet sich in Abschnitt A.1.

5.1.2 Compilerdirektiven

Die Anforderungen an die Software umfassten sowohl die Notwendigkeit einer großen Flexibilität, da sich die Plattform noch in der Entwicklung befand, als auch den Wunsch nach möglichst geringem Platzbedarf. Zu diesem Zweck existieren eine Reihe von Compilerdirektiven, die je nach Bedarf Funktionalitäten bereitstellen und nicht genutzte Codeteile deaktivieren. Der für die Bibliothek verwendete Compiler ist die *GNU-Compiler-Collection* `gcc`. Der Compiler wird vom GNU-Projekt als *Open-Source-Software*[2] entwickelt und ist für viele Kombinationen von Prozessoren und Betriebssystemen kostenlos verfügbar. Die verwendete Variante ist der `arm-uclibc-gcc`. Zur Vereinfachung des Kompilierungsvorgangs wurde ein weiteres GNU-Werkzeug verwendet, `make`. Mit dieser Software können Dateiabhängigkeiten automatisch überprüft werden. Dateien werden nur kompiliert, wenn sie sich seit dem letzten `make`-Aufruf verändert haben oder sie von einer veränderten Datei abhängen. Diese Abhängigkeitshierarchien werden in einer Konfigurationsdatei, dem *Makefile* aufgebaut. Dort lassen sich auch die Voreinstellungen für die im Folgenden beschriebenen Compilerdirektiven festlegen.

USE_AHB

Um die Unabhängigkeit der Software von der zugrunde liegenden Hardwareanbindung zu gewährleisten (DPSRAM oder AHB), existieren die meisten Funktionen in zwei Versionen. Die USE_AHB-Direktive bestimmt, welche der zwei Varianten verwendet wird. Wenn USE_AHB gesetzt ist, werden die Codeteile, die sich auf die DPSRAM-Anbindung beziehen, deaktiviert. Zusätzlich existieren zwei Modi für diese Direktive:

USE_AHB == 1 Die Software geht davon aus, dass die Resetleitung des AHB-Slaves an einen der Excalibur GPIO-Pins angeschlossen wurde und setzt bei Aufruf von *hpiReset* das GPIO-Register auf den übergebenen Wert `state`.

USE_AHB == 2 Die Software geht davon aus, dass die Resetleitung des AHB-Slaves vom CIAControl-Modul – einer zentralen Steuereinheit für die CIA-Karte – bedient wird und ruft die externe `setHPIBridgeReset`-Funktion auf, der `state`-Parameter wird dabei weitergereicht. Diese Einstellung ist im Makefile der Normalzustand.

[2] GNU ist ein rekursives Akronym und steht für GNU's Not Unix. Eine Einführung in das GNU-Projekt und die Konzepte der Open-Source-Software ist unter `http://www.gnu.org` zu finden.

Betroffene Funktionen:

hpiRead, *hpiStatus*, *hpiWrite*, *hpiReadMailbox*, *hpiWriteMailbox*, *hpiInit*, *hpiReset*, *unmapRegions*

USE_MAIN

Während der Entwicklungsphase der Bibliothek existierte eine `main`-Funktion, um die einzelnen Funktionsaufrufe schnell und unkompliziert testen zu können. Nach Fertigstellung der Bibliothek wurde `main` nicht entfernt, sondern als abhängiger Codeblock beibehalten. Wenn die Compilerdirektive USE_MAIN gesetzt ist, wird `main` in die Bibliothek eingebunden, sie kann dann als eigenständige Applikation aufgerufen werden. Die aktuelle Folge der Funktionsaufrufe dient der Aktivierung des EZ-Host als (funktionsfreies) USB-Peripheriegerät – damit kann getestet werden, ob die hardwareseitige Anbindung des Chips funktioniert. Natürlich kann `main` jederzeit um weitere Funktionalitäten erweitert werden.

Betroffene Funktionen:

keine

DEBUG

Mit dem Setzen der DEBUG-Direktive werden zusätzliche Statusmeldungen bei Funktionsaufrufen ausgegeben. Diese bestehen in den meisten Fällen aus einer Eintrittsmeldung und einer Austrittsmeldung[3], teilweise mit Ausgabe der Funktionsparameter. Relevante Laufzeitinformationen, wie etwa der Inhalt des GPIO-Registers vor und nach Schreibzugriffen, eventuell auftretende Fehler beim Öffnen des Speichergerätetreibers in *hpiInit* oder die verwendete Variante der Hardware-Anbindung werden ebenfalls ausgegeben.

Betroffene Funktionen:

hpiRead, *hpiStatus*, *hpiWrite*, *hpiReadMailbox*, *hpiWriteMailbox*, *hpiInit*, *hpiReset*, *unmapRegions*

BLOCK

Wird als Hardwareanbindung der AHB-Slave gewählt, so kann über die BLOCK[=0...3]-Direktive festgelegt werden, welchen PLD-Block der Slave verwendet. Die Konstanten PLD_BASE und PLD_SIZE werden entsprechend auf die in `stripe.h` definierten Werte für die Basisadresse bzw. die Adressbreite des gewählten PLD-Blocks gesetzt. *Wenn BLOCK ohne Zahlenangabe gesetzt wird, wird als Voreinstellung PLD-Block 2 gewählt.*

Betroffene Funktionen:

hpiInit, *unmapRegions*

[3] Beispiel für eine Eintrittsmeldung: `printf( ''Entering hpiReset'' );`.

5.1.3 Benutzung der Bibliothek

Auf der Bibliothek aufsetzende Applikationen können nach Bedarf auf die beschriebenen Funktionen zurückgreifen. Dabei sind einige Anforderungen zu beachten, die hier erläutert werden sollen. Bevor die Funktionen genutzt werden können, muss die Bibliothek initialisiert werden. Der entsprechende Funktionsaufruf ist `hpiInit()`. Das Betriebssystem erlaubt aus Sicherheitsgründen keine direkten Hardwarezugriffe. Der zur Hardware gehörige Adressbereich ist für den Benutzer nicht schreibbar. Um dem Programmierer dennoch eine Möglichkeit zu bieten, auf Geräte zuzugreifen, gibt es den `mmap`-Systemaufruf. Damit wird der verbotene Adressbereich in einen für das Programm zugänglichen Speicherbereich gespiegelt. In diesen Speicher geschriebene Daten werden dann an die reale Hardwareadresse weitergeleitet. Die Bibliotheksaufrufe `hpiRead()` und `hpiWrite()` verwenden Zeiger auf Adressbereiche, die zum Speichern der transferierten Daten verwendet werden. Diese Datenpuffer müssen zuvor angelegt werden. In den Funktionen findet keine Überwachung der Speichergrenzen statt. Werden also zu kleine Speicherbereiche alloziert, so wird der anschließende Speicherbereich überschrieben, was zu einer Zugriffsverletzung führen kann. Als letzter Bibliotheksaufruf sollte stets `unmapRegions()` ausgeführt werden. Er gibt die von hpiInit() gespiegelten Speicherbereiche wieder frei. Abbildung 5.1 zeigt ein Schema des Programmablaufs. Ein Beispiel ist der im Folgenden beschriebene Cypress Monitor CYMON.

5.2 Cypress Monitor (CYMON)

Zur Entwicklung der Firmware für den EZ-Host und als Beispiel für die Nutzung der ECCL-Bibliothek wurde der Cypress Monitor, CYMON, geschrieben. Er greift auf die Bibliotheksfunktionen zurück, ist aber deutlich einfacher zu bedienen. Als Kommandozeilenwerkzeug stehen die meisten Funktionen über Optionen zur Verfügung. Eine Zusammenfassung dieser Optionen liefert die *printUsage* Funktion. Es können mehrere Optionen in einem Aufruf angegeben werden, sie werden dann der Reihenfolge nach abgearbeitet. Dabei ist zu beachten, dass die Optionen Init (`-i`) und GoTo (`-g`) sich gegenseitig ausschließen. Die Continuous-Option (`-c`) führt, wie der Name sagt, in eine Schleife, die nur von außen unterbrochen werden kann. Sie sollte also als letzte Option angegeben werden. Eine typische Anwendung wäre, eine Firmware aus einer Datei in den Speicher zu laden, sie danach zu aktivieren und schließlich eine Statusabfrage durchzuführen, um zu überprüfen ob der Prozess ordungsgemäß gestartet wurde. Der zugehörige Aufruf sähe dann so aus:

```
cymon -u firmware.bin -g 04a4 -s
```

Bei einem Aufruf von CYMON wird zunächst die Schnittstelle initialisiert. Die Kommandozeilenparameter werden dann in einer Schleife über einen Aufruf der `getopt`-Systemfunktion abgefragt und anschließend in der *handleChoice*-Funktion abgearbeitet. Wenn keine Optionen mehr vorhanden sind, werden die Speicherbereiche wieder freigegeben und das Programm beendet sich. Das oben genannte Beispiel würde dann der Reihe nach die Funktionen *uploadFile*, *jumpToAddress* und die ECCL-Funktion *hpiStatus* aufrufen. Eine vollständige Funktionsreferenz findet sich in Abschnitt A.2. Im folgenden Abschnitt werden die vom CYMON erkannten Optionen erläutert.

5.2.1 Optionen

Der CYMON kennt neun Kommandozeilenoptionen. Wird er ohne Optionen aufgerufen, so wird eine knappe Meldung ausgegeben, die alle verfügbaren Optionen inklusive der zugehörigen Parameter auflistet. Es findet keine Plausibilitätskontrolle der Eingaben statt. Wird etwa als Adressparameter ein zu großer Wert angegeben, so ist das Ergebnis undefiniert. Die meisten Optionsparameter sind Hexadezimalwerte. Diese werden generell ohne Präfix eingegeben, also CE01, *nicht* 0xCE01.

upload `-u address file`

Schreibt Daten aus einer Datei in den Speicher des EZ-Host. Der `address`-Parameter bestimmt die Adresse, ab der in den Speicher geschrieben wird. Die Größe der Datei wird gelesen und als Parameter an den zugrunde liegenden *hpiWrite* Aufruf weitergegeben. Im Normalfall wird diese Funktion verwendet, um Firmware in den Speicher zu laden.

goto `-g address`

Springt an die im `address`-Parameter angegebene Speicherstelle und fährt dort mit der Programmausführung fort. Dies entspricht in der Wirkung dem Assembler JMP, wobei das Ziel stets eine absolute Adresse ist. Nach dem Laden einer Firmware-Binärdatei wird mit diesem Kommando deren `main`-Routine angesprungen. Die absolute Adresse der Routine wird durch das Linkerscript bestimmt. Sie ist entweder 0x04A4 (der Anfang des für Code verfügbaren Speichers) oder 0x1000, wenn im ersten Speichersegment Platz für den gdb-Stub[4] gelassen werden soll.

read `-r address length`

Liest `<length>` Datenwörter ab der Adresse `address` aus dem Speicher des EZ-Host und gibt sie in folgendem Format an `stdout`[5] aus:
@0xADDR: 0xDATA[6]
Die Ausgabe kann mit üblichen Unix-Mitteln in eine Datei umgeleitet werden. Diese Funktionalität ist in erster Linie für die Entwicklung von Software für den EZ-Host interessant und wird durch die Möglichkeiten des Debuggers weitgehend obsolet.

write `-w address length`

Liest `<length>` hexadezimale 16-Bit-Datenwörter von `stdin` und schreibt sie in den Speicher ab `address`. Auch diese Funktion ist vorwiegend für die Entwicklung relevant.

[4] Teil der Debugging-Technik, s. Abschnitt 6.5.
[5] Standardausgabegerät der Laufzeitumgebung, im Allgemeinen eine Kommandozeile. S. [CREF, S. 75f] für eine Einführung in das Konzept der Standardaus- und -eingabe (`stdin`).
[6] Im Quelltext: `printf("@0x%04x: 0x%04x", addr+(counter*2), g_data_p[counter]);`.

message `-m message`

Schreibt die angegebene Nachricht in das Mailboxregister des EZ-Host. Der `message`-Parameter ist ein präfixfreier Hexadezimalwert.[7]

init `-i state`

Ruft die ECCL-Funktion *hpiReset* mit dem übergebenen Parameter `state` auf. Der Parameter ist ein präfixfreier Hexadezimalwert. Welche Werte sinnvoll sind, hängt von der Hardwareimplementierung ab: da mit diesem Kommando der Zustand der Resetleitung verändert werden soll, muss der Wert bei Verwendung der Excalibur-GPIOs so gewählt sein, dass das richtige Bit im GPIO-Register gesetzt wird. Im Normalfall sollten allerdings die Werte 0 und 1 verwendet werden.

query `-q`

Liest das Mailboxregister des EZ-Host aus und gibt den Inhalt als hexadezimalen Wert auf `stdout` aus.

status `-s`

Liest das Statusregister des EZ-Host aus und gibt den Inhalt als hexadezimalen Wert auf `stdout` aus.

continuous `-c`

Die `continuous`-Option wurde eingeführt, um den Ablauf der Software auf dem EZ-Host kontinuierlich mitverfolgen zu können. In der konkreten Implementation werden in einer Schleife zunächst zwei Speicherstellen ausgelesen, die Zeiger auf die USB-Pufferspeicher enthalten (s. Kapitel 6). Danach wird ein Register überprüft, das von der Firmware modifiziert wird, wenn eine Anfrage über die USB-Schnittstelle eingeht. In diesem Fall werden die Puffer ausgelesen und nach `stdout` ausgegeben. Hierzu dienen die Funktionen *dumpCommand* und *dumpReply*. Um die Schleife zu beenden, muss der Speicher an der Adresse 0x3FFA mit dem Wert 0xDEAD beschrieben werden. Diese Speicherstelle wird als Abbruchbedingung der Schleife überprüft.

5.3 Einsatz in der Praxis

Sowohl die Bibliothek als auch das Kommunikationsmodul CYMON haben sich im Einsatz bewährt. Die Daten zur Messung des Durchsatzes des AHB-Slaves, die in Abbildung 4.5 dargestellt sind, wurden mit einem eigens geschriebenen Testprogramm erhoben, welches ebenfalls auf der ECCL aufsetzt. Abgesehen davon war der CYMON (und daher natürlich

[7] S. die von Cypress definierte Headerdatei `lcp_cmd.h` sowie [CBUM] für sinnvolle Nachrichten.

auch die ECCL) während der Entwicklung der Firmware für den EZ-Host in stetem Dauertest, etwa um neue Versionen in den Speicher zu laden oder kontinuierliche Laufzeitinformationen zu erhalten. Während der Entwicklung und Benutzung beobachtete Fehler sind behoben.

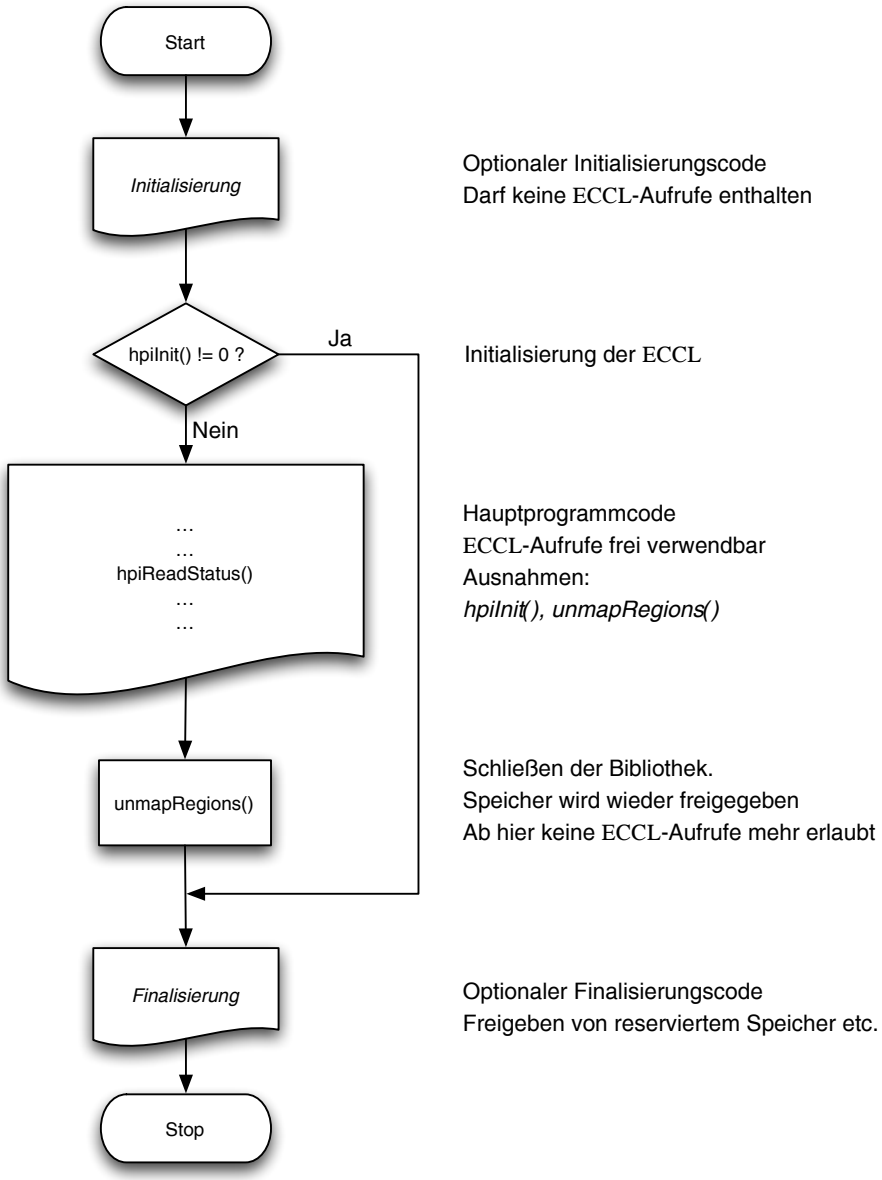


Abbildung 5.1: Ablaufschema eines ECCL-basierten Programmes

6 Entwicklung der Software für den EZ-Host

Dieser Teil der Arbeit befasst sich mit der Referenzimplementierung eines USB-basierten Massenspeichers. Hierzu wurde eine Firmware für den im Cypress EZ-Host enthaltenen CY16-Mikroprozessor geschrieben. Bei dem CY16 handelt es sich um einen 16-Bit-Prozessor mit USB-spezifischen Erweiterungen.[1] Um die Entwicklung von USB-Anwendungen zu erleichtern, wurde von der Firma Cypress ein Framework entwickelt, welches auf die grundlegenden BIOS-Funktionen zurückgreift. Des weiteren ist es möglich, über eine serielle Schnittstelle während des Betriebs Informationen über den Zustand des Chips zu erhalten. Dies kann entweder durch die Ausgabe von Fehlermeldungen oder über den GNU-Debugger `gdb` geschehen. Die zweite Variante bietet dabei mehr Möglichkeiten, wie in Abschnitt 6.5 beschrieben wird. Durch den Einsatz des Frameworks beschränkt sich die Programmierarbeit auf die Implementierung der Massenspeicherspezifikationen. Für die Implementierung von Massenspeichern stehen zwei Spezifikationen zur Verfügung, der Control/Bulk/Interrupt-Transport und der Bulk-Only-Transport. Allerdings wird in der Klassenspezifikation [MSSO] vom Einsatz des älteren Control/Bulk/Interrupt-Transport ausdrücklich abgeraten, dieser darf für neue Geräte nicht mehr verwendet werden. Die neuere MSBO-Spezifikation wurde daher als Basis für die Firmware verwendet.

6.1 USB-Massenspeicher

6.1.1 Die Mass-Storage-Class-Spezifikation

Eines der vordringlichsten Ziele bei der Erarbeitung der Spezifikation für USB-Massenspeicher war, die zahlreichen etablierten und bewährten Kommunikationsprotokolle einzubinden. Daher handelt es sich bei der *USB Mass Storage Class Specification* ([MSSO]) im Wesentlichen um eine USB-konforme Verpackung der bestehenden Protokolle. Einheitlich für alle Massenspeicher ist lediglich der Schnittstellenklassen-Code für Massenspeicher (0x08). Der Schnittstellensubklassen-Code legt das zugrunde liegende Kommunikationsprotokoll fest, in diesem Fall wurde das Universal-Floppy-Interface (UFI, Code 0x04) gewählt. Der Schnittstellenprotokoll-Code schließlich bestimmt das gewählte Transport-Protokoll, hier also der Bulk-Only-Transport (0x50).

6.1.2 Mass-Storage-Bulk-Only-Transport (MSBO)

Grundlage für die Implementierung des Massenspeichers war die von der Mass Storage Class Specification Workgroup herausgegebene *Mass Storage Bulk Only* (MSBO)-Spezifi-

[1] Die Architektur des Prozessors wird in [CY16] umfassend beschrieben.

kation ([MSBO]). Bei diesem Protokoll werden sowohl Nutzdaten als auch Kommandos über zwei Bulk-Endpunkte[2] mit dem Host ausgetauscht. Der obligatorische Control-EP wird nur zum Austausch der Deskriptoren bzw. zum Zurücksetzen des Gerätes verwendet.[3] Wie bereits erwähnt, handelt es sich im Wesentlichen um einen Verpackung des zugrunde liegenden – und als Industriestandard etablierten – Kommunikationsprotokolls. Diese Funktionalität wird über zwei Datenstrukturen erreicht, den *Command Block Wrapper* (CBW) sowie den *Command Status Wrapper* (CSW). Der Host sendet zunächst einen CBW, welcher das vom Slave auszuführende Kommando enthält. Anschließend folgen die für die Ausführung eventuell notwendigen Daten.[4] Der Transfer wird durch einen vom Slave versendeten CSW abgeschlossen. Jeder CBW besitzt einen zugehörigen CSW, die Zuordnung erfolgt über ein gemeinsames, vom Host für jeden CBW festgelegtes Datenwort.

CBW Der CBW besteht aus einer 31 Byte langen Datenstruktur. Tabelle 6.1 gibt einen Überblick über die Anordnung. Die ersten vier Bytes enthalten die die Signatur. Diese ist für alle CBW-Pakete identisch 0x43425355. Die folgenden vier Bytes enthalten ein für jeden CBW unterschiedliches Datenwort[5], dieser wird von dem korrespondierenden CSW reflektiert. Die nächsten vier Bytes geben an, wieviele Bytes der Host als Antwort auf das aktuelle Kommando erwartet. Ist dieser Wert 0, so sollten keine Daten zwischen CBW und CSW ausgetauscht werden. Es folgt ein Flagbyte. Relevant ist nur das höchste Bit, es bestimmt die Richtung des Datentransfers, der aus dem Kommando resultiert. 0 bedeutet, der Host sendet Daten an das Gerät. 1 bedeutet, der Host erwartet Daten von dem Gerät. Das folgende Byte 13 beinhaltet die Gerätenummer (LUN), für die das Kommando gelten soll. Unterstützt das Gerät keine mehrfachen LUNs[6], so wird der Host in diesem Byte 0 senden. Byte 14 enthält die Länge des darauf folgenden Kommandowortes. Der Slave muss Daten, die über die hier angegebene Länge hinausgehen, ignorieren. Die Bytes 15 – 30 enthalten den eigentlichen Kommandoblock. Dieser ist immer 16 Byte lang, unabhängig von der eigentlichen Länge des Kommandos.

CSW Der CSW ist 13 Bytes lang. Die ersten vier Bytes enthalten die CSW-Signatur, für alle CSW-Pakete identisch 0x53425355. Die folgenden vier Bytes bestehen aus dem Identifikations-Datenwort des vorausgegangenen CBW. Produziert die Ausführung des Kommandos weniger Daten, als der Host als Antwort erwartet, so wird die Differenz zwischen angeforderten und vorhandenen Daten in den folgenden vier Bytes mitgeteilt. Das letzte Byte enthält den Status, mit dem das ausgeführte Kommando abgeschlossen wurde. Erlaubt sind die in Tabelle 6.2 angegebenen Werte.

[2] S. Abschnitt 2.2

[3] Dieser Teil wird vollständig von dem BIOS des EZ-Host-Chips abgewickelt, ist also nicht Teil dieser Arbeit.

[4] Bei einem Schreibkommando z. B. die zu schreibenden Daten.

[5] In Treibern meist ein linear inkrementierter Zähler, dies ist in der Spezifikation aber nicht festgelegt, sondern implementationsabhängig.

[6] Diese Information erhält der Host über den Klassen-Request *GetMaxLUN*. In der vorliegenden Implementierung werden keine LUNs unterstützt.

(a) Aufbau des Command Block Wrappers

Byte	Beschreibung
0 – 3	CBWSignature (0x43425355, little endian)
4 – 7	CBWTag
8 – 11	CBWDataTransferLength
12	CBWFlags
13	CBWLUN (Bits 0 – 3)
14	CBWBLength (Bits 0 – 4)
15 – 30	CBWCB (enthalten in Bytes 15 bis 15 + CBWBLength)

(b) Aufbau des Command Status Wrappers

Byte	Beschreibung
0 – 3	CSWSignature (0x53425355, little endian)
4 – 7	CSWTag (aus dem vorangehenden CBWTag)
8 – 11	CSWDataResidue
12	CSWStatus

Tabelle 6.1: Aufbau der CBW- (a) und der CSW-Datenstruktur (b), Quelle: [MSBO]

Wert	Beschreibung
0x00	Command Passed (Status GOOD)
0x01	Command Failed
0x02	Phase Error

Tabelle 6.2: Statuswerte für den Command Status Wrapper, Quelle: [MSBO]

6.1.3 Universal-Floppy-Interface (UFI)

Für das zugrunde liegende Kommunikationsprotokoll fiel die Wahl auf das *Universal Floppy Interface* ([UFI]). Das Protokoll umfasst die im Folgenden aufgeführten 19 Kommandos. Die einzelnen Kommandoblöcke sind jeweils 12 Bytes lang, die zugehörige Datenstruktur folgt dem in Tabelle 6.3 angegebenen Schema. Die Kommandoblöcke sind im CBWCB-Feld untergebracht.

Format Unit Unformatierte Medien formatieren. Diese Operation wird von der in dieser Arbeit implementierten Firmware nicht abgedeckt, da keine Schreibzugriffe auf das Gerät erlaubt sind.

Inquiry Information über das Gerät anfragen. Die Firmware antwortet mit einem statischen einkompilierten Datensatz.

Start / Stop Unit Aktiviert / Deaktiviert den Zugriff auf ein eingelegtes Medium. Wird von der Firmware ignoriert und mit einem „Command passed“ beantwortet.

Mode Select Erlaubt dem Host, Parameter im Gerät zu verändern. Ist in der Firmware

derzeit nicht implementiert.

Mode Sense Erlaubt dem Gerät, Zustandsinformationen an den Host zu übermitteln. Ist in der Firmware derzeit nicht implementiert.

Prevent / Allow Medium Removal Verhindert / Erlaubt das Entfernen eines eingelegten Mediums in das Gerät. Wird von der Firmware ignoriert, da keine Wechselmedien vorgesehen sind.

Read(10) Liest Daten vom Medium.

Read(12) Liest Daten vom Medium. Diese Variante vergrößert die maximale Transferlänge auf bis zu 2^{32} Blöcke.

Read Capacity Liefert dem Host die maximale Kapazität des eingelegten Mediums. Wird von der Firmware mit einem statisch einkompilierten Datensatz beantwortet.

Read Format Capacity Liefert dem Host eine Liste der vom Gerät unterstützten Formatkapazitäten. Wird von der Firmware mit einem statisch einkompilierten Datensatz beantwortet.

Request Sense Liefert dem Host erweiterte Informationen zum Status des letzten Kommandos. Ist in der Firmware derzeit nicht implementiert.

Rezero Unit Setzt den Lesekopf auf Position 0. Wird von der Firmware ignoriert, da kein Lesekopf vorhanden ist. Dieses Kommando könnte in einen `rewind`-Systemaufruf auf der Excaliburseite umgesetzt werden, um die Antwortzeiten zu optimieren.

Seek Bewegt den Lesekopf an die angegebene Adresse. Wird von der Firmware ignoriert, da kein Lesekopf vorhanden ist. Dieses Kommando könnte in einen `seek`-Systemaufruf auf der Excaliburseite umgesetzt werden, um die Antwortzeiten zu optimieren.

Send Diagnostic Setzt das Gerät zurück und führt eine Diagnose durch. Ist in der Firmware derzeit nicht implementiert.

Test Unit Ready Fragt die Bereitschaft des Gerätes ab. Die Firmware gibt bei dieser Anfrage stets einen Status GOOD zurück.

Verify Verifiziert die Daten auf dem Medium. Ist in der Firmware derzeit nicht implementiert.

Write(10) Schreibt Daten auf das Medium. Ist in der Firmware derzeit nicht implementiert.

Write(12) Schreibt Daten auf das Medium. Diese Variante vergrößert die maximale Transferlänge auf bis zu 2^{32} Blöcke. Ist in der Firmware derzeit nicht implementiert.

Write and Verify Schreibt Daten auf das Medium und verifiziert sie anschließend. Ist in der Firmware derzeit nicht implementiert. Alle Schreibaufrufe sollten von der Firmware mit einem Fehler beantwortet werden, da Schreibzugriffe nicht erlaubt sind (Read-Only).

Byte	Belegung
0	Kommandocode
1	LUN (Bits 5 – 7) / Reserviert (Bits 0 – 4)
2 – 5	Logische Blockadresse
6	Reserviert
7 – 8	Transferliste, Parameterliste oder Datenlänge
9 – 11	Reserviert

Tabelle 6.3: Schema der UFI-Kommandostrukturen

6.2 BIOS des EZ-Host-Chips

Das BIOS des EZ-Host verfügt über eine Reihe von Routinen zur Abarbeitung von Unterbrechungen (ISR), die der Programmierer direkt aufrufen kann. Diese sind in [CBUM] umfangreich beschrieben. Der EZ-Host wird in der vorliegenden Implementierung im Koprozessormodus betrieben.

6.2.1 Link-Control-Protocol (LCP)

Die Funktionen des Chips werden über das *Link Control Protocol* zugänglich, welches vollen Zugriff auf die Interna des Chips erlaubt. Wird über HPI ein LCP-Kommando an den EZ-Host geschickt, so wird es in einem dedizierten COMM_PORT_CMD-Register abgelegt. Dieses wird periodisch von der Firmware überprüft, welche das gesendete Kommando ausführt und den Rückgabewert in das Mailboxregister schreibt. Dort kann sie dann über einen entsprechenden HPI-Aufruf ausgelesen werden. Das LCP umfasst die folgenden 11 Kommandos:

COMM_RESET Setzt den `lcp_idle`-Task zurück.

COMM_JUMP2CODE Springt an die angegebene Adresse und führt den dort abgelegten Code aus.

COMM_CALL_CODE Ist identisch mit JUMP2CODE bis auf ein Detail: Diese Funktion sendet ein COMM_ACK erst *nachdem* der Code ausgeführt wurde. JUMP2CODE sendet *zuerst* ein COMM_ACK und führt danach den Code aus.

COMM_EXEC_INT Löst die angegebenen ISR aus.

COMM_READ_CTRL_REG Liest Daten aus dem angegebenen Kontrollregister. Der direkte Speicherzugriff (DMA) des HPI-Protokolls schließt den Speicherbereich zwischen 0xC000 und 0xC0FF aus Sicherheitsgründen aus. Um trotzdem Zugriff auf die dort angelegten Kontrollregister zu erhalten, werden dieses und das folgende Kommando verwendet.

COMM_WRITE_CTRL_REG Schreibt Daten in das angegebene Kontrollregister.

COMM_READ_MEM Liest aus dem internen Speicher des EZ-Host. Wird wegen des DMA unter HPI nicht benötigt.

COMM_WRITE_MEM Schreibt in den internen Speicher des EZ-Host.

COMM_READ_XMEM Liest Daten aus einen extern an den EZ-Host angeschlossenen Speicher. Wird nicht verwendet, da kein externer Speicher angeschlossen ist.

COMM_WRITE_XMEM Schreibt Daten in einen extern an den EZ-Host angeschlossenen Speicher.

COMM_CONFIG Ändert die Konfiguration der seriellen Schnittstelle. Wird unter HPI nicht benötigt.

Der Programmierer erhält über die Mailbox der HPI-Schnittstelle Zugriff auf die LCP-Kommandos. Die CYMON-Funktion *jumpToAddress* z. B. verwendet das LCP-Kommando COMM_JUMP2CODE.

6.3 Framework

Das EZ-Host-Framework enthebt den Firmwareprogrammierer von der Notwendigkeit, alle Details des USB-Protokolls selbst zu implementieren, indem es die unteren Protokollschichten wie z. B. die Standard-Requests automatisch abarbeitet. Das Framework stellt auch zahlreiche Hilfsfunktionen und Makros zur Verfügung. Beispielsweise Wrapperfunktionen, die eine einfache C-Schnittstelle zu den im BIOS definierten ISRs bieten. Datenstrukturen zur Beschreibung der USB-Deskriptoren sind bereits vordefiniert und müssen nur mit konkreten, zur Anwendung passenden Werten ausgefüllt werden. Eine Referenz der im Framework definierten Funktionen und Makros ist in [CFRG] gegeben. Der Entwickler kann sich bei der Programmierung auf den anwendungsrelevanten Teil beschränken. Er konfiguriert das Framework und implementiert dann die erforderlichen Subroutinen in C oder Assembler. Die Konfiguration geschieht durch das Anpassen der Konfigurationsdatei `fwxcfg.h`, in der über Compilerdirektiven gezielt Funktionen des Frameworks ein- oder ausgeschlossen werden können. Der EZ-Host ist in der Lage, sowohl als Peripheriegerät als auch als Host zu arbeiten. Zudem unterstützt er die OTG-Erweiterung der USB 2.0 Spezifikation. Für alle Hardwarefunktionen existiert auch Softwareunterstützung im Framework. Abbildung 6.1 zeigt ein Ablaufschema für ein reines Peripheriegerät. Nach der Initialisierungsphase wird eine Endlosschleife durchlaufen. Da der EZ-Host keine Nebenläufigkeit beherrscht, werden in der Schleife verschiedene Idle-Funktionen aufgerufen, die unterschiedliche Aufgaben bearbeiten. So kann z. B. der Debugger-Stub Daten mit dem `gdb` austauschen. Wird durch Aktivität an der USB-Schnittstelle eine Unterbrechung ausgelöst, so wird diese abgearbeitet. Danach kehrt das Programm in die Schleife zurück.

6.3.1 Erläuterung der Framework-Funktionen

In diesem Abschnitt werden die vom Framework in der Hauptschleife durchlaufenen Subroutinen kurz dargestellt. Eine ausführliche Beschreibung ist in [CFRG] zu finden.

bios_idle Ruft den BIOS-Idle-Task auf. In diesem werden die für den Ablauf der Basisfunktionalität des Chips notwendigen Routinen `usb_idle`, `lcp_idle` und `uart_idle`

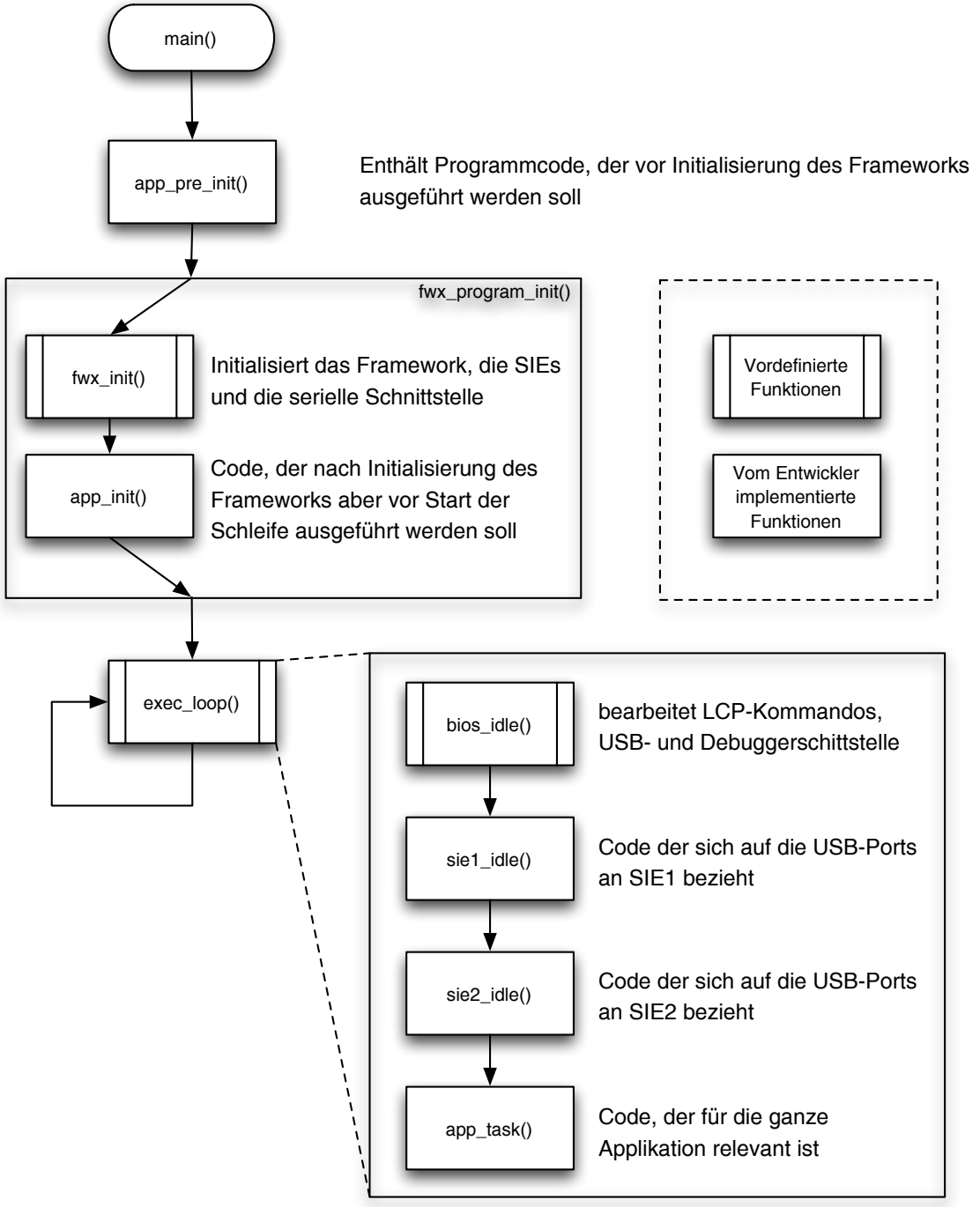


Abbildung 6.1: Flussdiagramm des EZ-Host Frameworks

aufgerufen. Diese Routinen verwalten die Kommunikationsressourcen des EZ-Host-Chips. Die `usb_idle`-Routine bearbeitet USB-Ereignisse. Die `lcp_idle`-Routine liest das LCP-Kommandoregister und führt die dort ankommenden Kommandos aus. Die `uart_idle`-Routine verwaltet die vom Debugger verwendete serielle Schnittstelle. Die `bios_idle`-Funktion wird nur aufgerufen, wenn in der Konfigurationsdatei die Option FWX_SERIAL_EEPROM nicht definiert ist. Dies ist automatisch der Fall, wenn die Firmware für das Debugging kompiliert wird. Ansonsten wird der BIOS-Idle-Task von der `idler_chain` aufgerufen.

sie1_idle Diese Funktion behandelt Ereignisse, die die SIE1 betreffen. Sie wird vom Firmwareprogrammierer implementiert. In der implementierten Firmware wird in dieser Funktion der größte Teil der Arbeit durchgeführt.

app_task In dieser Funktion werden Aufgaben bearbeitet, die für die gesamte Firmware relevant sind. Der Parameter dieser Funktion beschreibt den aktuellen Laufzustand der Firmware. In der implementierten Firmware wird der Parameter abgefragt und je nach Zustand der Wert 0xDEAD (System angehalten) oder 0xBEEF (System läuft) in das Register 0x3FF6 geschrieben. Dieses Register kann dann vom Excalibur-Prozessor zur Zustandsüberwachung der EZ-Host-Firmware ausgelesen werden.

6.4 Implementierung der Firmware

6.4.1 Entwicklungsumgebung

Für den EZ-Host existiert eine Entwicklungsumgebung. Sie enthält neben der Dokumentation, den in [CFRG] beschriebenen Framework-Bibliotheken und Headerdateien auch eine an den CY16-Mikroprozessorkern des Chips angepasste Variante des `gcc`-Compilers, den `cy16-elf-gcc`. Das `make`-Werkzeug ist ebenfalls Bestandteil der Umgebung. Für frameworkbasierte Anwendungen liefert die Firma Cypress ein Makefile mit passenden Voreinstellungen, das noch an das konkrete Projekt angeglichen werden muss. Im Makefile existieren zwei Entwicklungszweige (Targets). Das Release-Target hat die Kompilierung eines möglichst kleinen und effizienten Programmes zum Ziel. Dazu werden alle für die Ausführung nicht notwendigen Teile des Codes entfernt. Das Debug-Target hingegen bindet zum Debugging wichtige Informationen in die ausführbare Binärdatei. Das Programm wird dadurch größer. Zudem wird im Linkerskript, in dem die Reihenfolge festgelegt wird, in der die einzelnen Codesegmente verbunden werden, die Startadresse der `main`-Routine von 0x04A4 auf 0x1000 verschoben. Dies geschieht, um Platz für den vom Debugger benötigten Stub zu schaffen. Dies ist beim Start der Firmware, z. B. über die *goto*-Option des CYMON, zu berücksichtigen

6.4.2 Funktionsprinzip

Das Funktionsprinzip der implementierten Firmware ist in Abbildung 6.2 abgebildet.

In der `app_init`-Funktion wird lediglich das Register 0x3FF6 auf Null initialisiert. Durch Auslesen dieses Registers über den CYMON kann der Programmierer den Laufzustand fest-

stellen. Das Hostbetriebssystem muss zunächst über einen *SetConfiguration*-Request eine Geräte-Konfiguration auswählen. Die Firmware erhält dabei eine Unterbrechung, die von der `susb1_delta_cfg_handler`-Funktion abgearbeitet wird. Diese initialisiert eine Zustandsvariable `g_transferState` und startet damit die eigentliche Anwendung. Nach dem Abschluss der Konfiguration wartet das Gerät auf Anfragen vom Host. Dazu ruft es die `susb_receive`-Funktion auf und überprüft auf dem Bulk-In-EP ankommende Daten auf die CBW-Signatur. Wird ein Paket als CBW erkannt, so wird zunächst überprüft, ob der Host weitere Daten senden will oder erwartet. Folgen keine Daten, so wird die Paketkennung CBWTag in den zugehörigen CSW kopiert und anschließend das übertragene Kommando ausgeführt. Der Rückgabewert wird in das CSWStatus-Feld eingesetzt und der CSW über einen Aufruf der Frameworkfunktion `susb_send` über den USB an den Host übertragen. Sieht der Host Daten für den Transfer vor, so wird zunächst die Transferrichtung festgestellt. Entsprechend des Richtungsbits im CBWFlag-Feld wird der EZ-Host durch einen Aufruf der `susb_send` bzw. `susb_receive`-Funktion auf den Versand bzw. den Empfang von Daten eingestellt. Diese Funktionen erwarten drei Parameter. Der erste enthält die verwendete SIE. In der implementierten Firmware wird die SIE1 verwendet. Der zweite Parameter beinhaltet den für den Transfer verwendeten Endpunkt. Für `susb_send` ist das der Bulk-In-EP2, für `susb_receive` der Bulk-Out-EP1. Der dritte Parameter ist ein Zeiger auf eine Datenstruktur, die Informationen zum Transfer enthält. Das erste Feld dieser Struktur ist für künftige BIOS-Erweiterungen reserviert. Das zweite Feld enthält einen Zeiger auf einen Transferpuffer, der zum Transfer der Daten verwendet wird. Das dritte Feld enthält die Größe dieser Datenpuffers in Byte. Das letzte Feld beinhaltet einen Zeiger auf eine Funktion, die über eine Unterbrechung aufgerufen wird wenn der Transfer beendet ist. Um die Abarbeitung dieser Unterbrechungs-Routine, `xfer_done`, so kurz wie möglich zu machen, werden lediglich die unmittelbar notwendigen Aktionen – Überprüfung des CBW oder der Transferlänge – direkt ausgeführt. Danach wird die Zustandsvariable gesetzt. Diese wird dann beim nächsten Aufruf der `sie1_idle`-Funktion abgearbeitet. Dies geschieht auch beim Versand oder Empfang von Nutzdaten. Zuvor wird in der `execute_command`-Funktion das UFI-Kommando auf dem CBWCB-Feld extrahiert und ausgeführt. Das Ergebnis der Kommandoausführung wird im Rückgabewert der Funktion propagiert und über das CSWStatus-Feld an den Host geleitet. Danach wird die Zustandsvariable wieder auf den Ausgangswert zurückgesetzt.

6.5 Debugger

Ein weiteres GNU-Werkzeug, der Debugger `gdb`, ermöglicht den Zugriff auf den internen Ablauf der Firmware und gibt Einblick in die Register und den Speicher. Der EZ-Host enthält eine serielle Schnittstelle, die, bei entsprechenden Einstellungen in der Konfigurationsdatei fwxcfg.h, zur Kommunikation mit dem Debugger verwendet wird. Über die von Cypress mitgelieferte graphische `gdb`-Anwendung Insight wird zunächst eine Verbindung mit dem EZ-Host hergestellt. Danach wird ein Debugger-Stub, eine kompakte Kommunikationssoftware für den `gdb`, in den unteren Bereich[7] des internen Speichers geladen. Anschließend wird der eigentliche Firmware-Code in den Speicher übertragen.[8] In Insight

[7] Adresse 0x04A4 – 0x1000, der erste Block des frei nutzbaren Speicherbereichs.

können dann Markierungen in den Quellcode gesetzt werden, um an der gewünschten Stelle in den Programmablauf einzugreifen. Der EZ-Host ist nicht multitaskingfähig, daher stellt Nebenläufigkeit hier kein Problem dar. Allerdings werden die „interessanten“ Ereignisse durch Unterbrechungen ausgelöst, hier ist der Debugger nur begrenzt einsatzfähig.

6.6 Entwicklungsstand

Gegenwärtig wird das Gerät vom Betriebssystem als Massenspeicher erkannt und als Laufwerk mit eigener Laufwerkskennung angezeigt. Ein Teil der UFI-Kommandos ist bereits implementiert. Lesezugriffe werden von der Firmware unterstützt, es existiert aber noch keine Datenquelle auf Seite des Excalibur-Bausteins. Auf mögliche Lösungen wird im nächsten Kapitel eingegangen. Das INQUIRY-Kommando zur Abfrage des Gerätezustandes ist ebenfalls funktional. Ebenso die Anfragen nach der Kapazität des Mediums, READ CAPACITY und READ FORMAT CAPACITY. Diese Informationen sind statisch in die Firmware einkompiliert. Schreibzugriffe werden nicht unterstützt. Die geplante Verwendung des Massenspeichers z. B. um den Host mit neuen BIOS-Versionen zu versehen benötigt nur Lesezugriffe. Daher wurden die für Schreibzugriffe notwendigen UFI-Kommandos in der ersten Fassung der Firmware nicht vorgesehen.

Die konkrete Umsetzung der Massenspeicherspezifikationen wird von Hostbetriebssystemen unterschiedlich gehandhabt. Während der Arbeit wurde der Massenspeicher überwiegend unter Microsoft Windows 2000 getestet. Dazu werden zwei Programme verwendet:

USB Command Verifier Dies ist das offizielle Programm für Kompatibilitätstests des *USB Implementers Forums* (USB-IF), dem USB-Standardisierungsgremium. Zu beziehen unter `http://www.usb.org`.

SnoopyPro Ein quelloffenes Programm, das den Datenverkehr an der USB-Schnittstelle protokolliert. Die Datenpakete können anschließend gesichtet und analysiert werden. Zu beziehen unter `http://sourceforge.net/projects/usbsnoop`.

Der fertige Massenspeicher wird aber durch die konsequente Einhaltung der Spezifikationen mit jedem Betriebssystem verwendbar sein.

[8] Hier ist zu beachten, dass dem Compiler beim Kompilieren der Firmware im Linkerscript mitgeteilt wird, dass die Einsprungadresse sich verschiebt, um Platz für den Debugger-Stub zu lassen, s. Unterabschnitt 6.4.1.

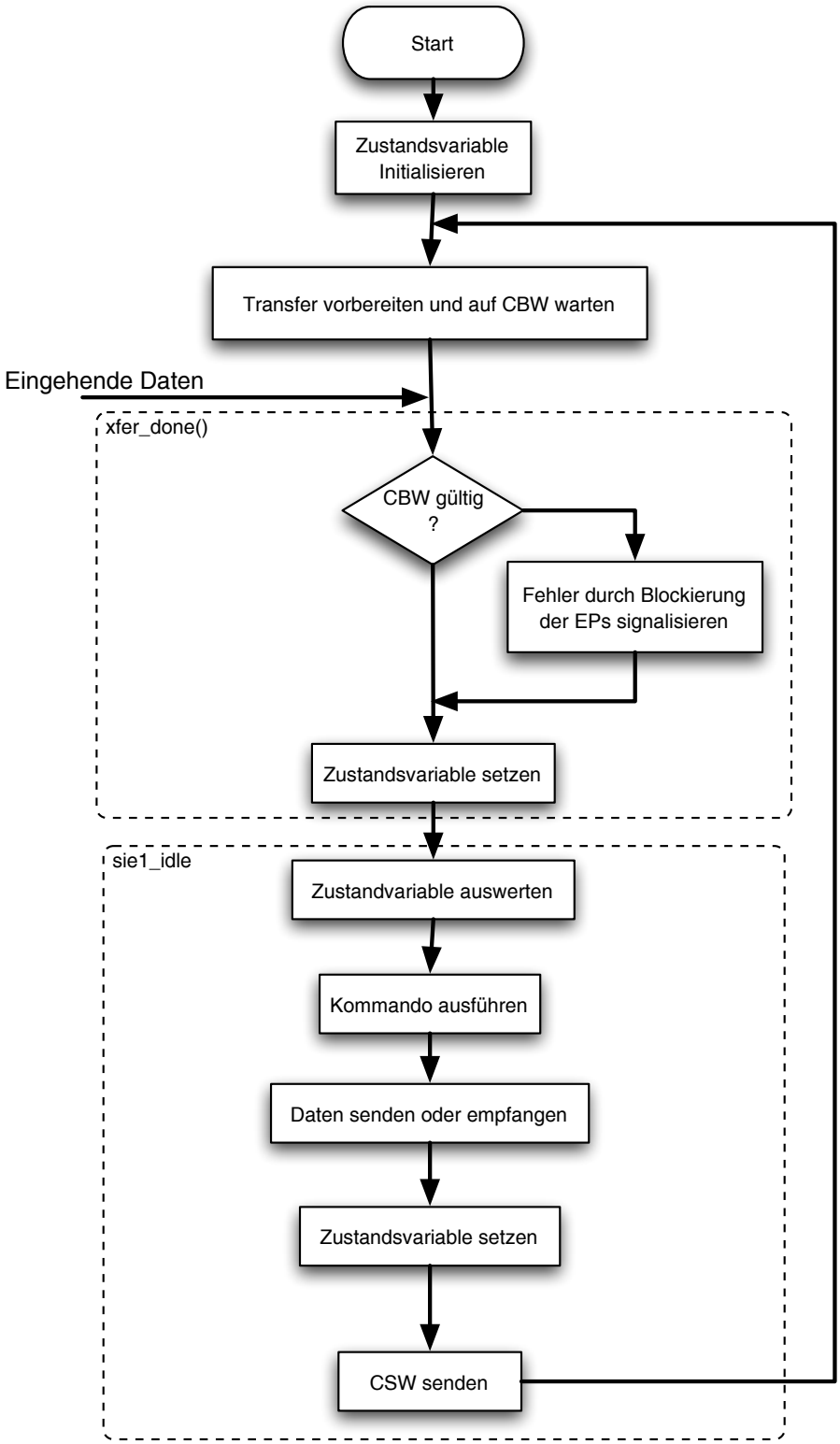


Abbildung 6.2: Flussdiagramm der implementierten Firmware

7 Zusammenfassung und Ausblick

Diese Arbeit beschreibt die Einbindung einer USB-Schnittstelle in ein kompaktes Diagnose- und Fernwartungssystem auf Basis einer PCI-Erweiterungskarte. Zwei PLD-basierte Entwürfe zur Hardwareanbindung des Schnittstellenchips werden vorgestellt und verglichen. Der AHB-Slave erweist sich dabei als flexibler und effizienter als der Zugriff über die Speicherschnittstelle. Der Entwurf ist gut portabel. er kann also mit wenig Aufwand auf andere Systeme übertragen werden. Die zur Ansteuerung der Hardware entwickelte Funktionsbibliothek ist ausführlich dokumentiert. Dem Programmierer wird damit ein einfaches und effizientes Mittel zur Verfügung gestellt, um spezialisierte Anwendungen zur Kommunikation mit der Hardware zu entwickeln. Dem Anwender steht mit dem vorgestellten CYMON ein auf der Funktionsbibliothek aufsetzendes Werkzeug zur Verfügung, um alle Informationen über den Zustand des EZ-Host-Chips abzufragen. Die Hardwareanbindung ist getestet und hat sich im Entwicklungseinsatz bewährt. Auch die Kommunikationssoftware sowie das Beispielprogramm zeigen in Tests keine Fehler. Die Einsatz als realer Massenspeicher konnte mangels einer vollständigen Firmware für den EZ-Host nicht getestet werden. Das Rahmenprogramm der Firmware ist aber gegeben und funktional. Die endgültige Implementierung des Massenspeichers sich damit im Wesentlichen auf die Vervollständigung der von der Firmware erkannten UFI-Kommandos beschränken. Der von der implementierten Schnittstelle erzielte Datendurchsatz ist für die Anwendung mehr als ausreichend.

Vordringlichstes Ziel einer Weiterentwicklung wird die vollständige Implementierung der Massenspeicherspezifikationen in der Firmware des EZ-Host-Chips sein. In einer zweiten Fassung kann dabei auch die bisher bewusst unberücksichtigt gelassene Schreibfunktionalität realisiert werden. Zuvor muss die noch offene Frage der Datenquelle gelöst werden. Der über die *continuous*-Option im CYMON implementierte Dauerüberwachungsmechanismus kann erweitert werden, um diese Aufgabe zu übernehmen. Dazu müssen Adresse und Länge der vom Host geforderten Daten von der Firmware in ein Register geschrieben werden. Diese werden dann vom CYMON in regelmäßigen Abständen aus dem Register ausgelesen. Wird eine Datenanfrage festgestellt, so werden die geforderten Daten in einen Pufferspeicher geschrieben, wo sie von der Firmware an den Host übertragen werden. Allerdings wäre eine unterbrechungsgesteuerte Variante effizienter. In diesem Fall würde die Firmware bei einer Hostanforderung eine Unterbrechung im Excalibur auslösen, die dieser dann abarbeitet. Die Unterbrechungsleitung ist im Entwurf des AHB-Slaves bereits enthalten. Dazu müsste ein geeigneter Treiber für das CIA2-Betriebssystem geschrieben werden. Das hätte auch den Vorteil, das Zugriffe auf die Hardware nicht mehr über Spiegeln des AHB-Slave-Adressraums stattfinden müssen. So können wertvolle Speicherressourcen gespart werden.

Der Datendurchsatz des AHB-Slaves ist auch für Full-Speed-Anwendungen ausreichend. Trotzdem wäre eine Steigerung des Durchsatzes möglich, indem die sowohl im AHB- als auch im HPI-Protokoll vorgesehenen Mechanismen für Daten-Bursts ausgenutzt werden. Da der EZ-Host das Adressregister nach einem Zugriff automatisch erhöht, könnten

aufeinander folgende Daten ohne zusätzlichen HPI-Adresszyklus übertragen werden. Der Durchsatz könnte so nahezu verdoppelt werden. Eine weitere Erweiterungsmöglichkeit des AHB-Slaves betrifft die Frequenzabhängigkeit. Die aktuelle Implementierung erwartet eine Taktfrequenz von höchstens 40 MHz. Diese Taktung wird in der CIA2-Karte für AHB-Komponenten verwendet. Ist der Takt zu schnell, würden die Timing-Anforderungen des HPI-Protokolls nicht mehr erfüllt. Fehler bei der Datenübertragung wären die Folge. Eine mögliche Lösung sind konfigurierbare Wartezyklen. Die aktuelle Fassung sieht bereits zwei solche Wartezustände vor, XFER_PAUSE und XFER_FINISH. Diese sichern die benötigten sechs EZ-Host-Takte Abstand zwischen zwei HPI-Zyklen. Diese Wartezyklen könnten durch Einsatz eines Taktzählers um die zur Protokollwahrung erforderliche Anzahl an Takten verlängert werden. Der AHB-Slave wäre durch diese Erweiterung auch bei höheren Taktraten einsetzbar.

Anhang A

Softwaredokumentation

A.1 ECCL-Funktionsreferenz

getMappedHPIBase()

Überblick Liefert einen Zeiger auf den Adressraum der HPI-Schnittstelle.

Details Der obligatorische Aufruf von *hpiInit()* spiegelt den (statischen) Adressraum der HPI-Schnittstelle in den Adressraum des Programmes. Bis zum Aufruf von *unmapRegions()* bleibt der Zeiger auf diesen Speicher erhalten, er kann mittels dieser Funktion abgerufen werden.

Parameter keine

Rückgabewert `uint16*` : Zeiger auf die Basis des gespiegelten Adressraums der HPI-Schnittstelle

getMappedRegisterBase()

Überblick Liefert einen Zeiger auf einen Speicherbereich, in den die Excalibur-Register gespiegelt werden.

Details Um auf das GPIO-Register des Excalibur-Chips zugreifen zu können[a], müssen alle Register in den Adressraum des Programmes gespiegelt werden. Das GPIO-Register findet sich dann an Offset 0x150.[b]

Parameter keine

Rückgabewert `uint32*` : Zeiger auf die Basis des Speicherbereichs, in den die Excalibur-Register gespiegelt werden.

[a] Diese werden vor Allem bei Verwendung der DPSRAM-Schnittstelle als Steuersignale eingesetzt, aber auch, um den AHB-Slave zurückzusetzen.

[b] Beispiel: `*(g_gpio_p + 0x54) = state;`

hpiInit()

Überblick Initialisiert die Bibliothek. Diese Funktion muss vor allen anderen Bibliotheksfunktionen aufgerufen werden.

Details Zunächst wird eine Verbindung zum Speichergerätetreiber (`/dev/mem`) geöffnet.[a] Danach werden mittels des Linux-Funktionsaufrufs `mmap` die Adressbereiche der verwendeten Hardwareanbindung (DPSRAM oder AHB, s. Unterabschnitt 5.1.2) sowie der Excalibur-Register in den Adressraum des Programmes gespiegelt. Zuletzt wird die Verbindung zum Speicher wieder geschlossen, die Zeiger auf Register und HPI-Schnittstelle bleiben weiter gültig und können mit den Funktionen *getMappedRegisterBase* bzw. *getMappedHPIBase* abgefragt werden.

Parameter keine

Rückgabewert `int`: `errno` falls ein Fehler auftritt, 0 sonst.

[a] Sollte dies fehlschlagen, so wird eine Fehlermeldung ausgegeben und das Programm bricht mit `exit(-1)` ab.

hpiRead(addr, data, length)

Überblick Liest Daten aus dem internen Speicher der EZ-Host-Chips.

Details Auch diese Funktion hat zwei Implementierungen, je nach Wahl der Compilerdirektive USE_AHB. Während die DPSRAM-Variante deutlich mehr Aufwand betreiben muss, um die Timing-Anforderungen des HPI-Protokolls zu wahren[a], erfolgt dies in der AHB-Variante implizit im Inneren des PLDs. Folgerichtig ist die zugehörige Implementierung im Wesentlichen eine Schleife, die dem `data`-Puffer den Inhalt des angeforderten Speicherbereichs zuweist. Zusätzlich findet eine Überprüfung des `length`-Parameters statt. Wenn die Anzahl der Wörter die Speichergröße des EZ-Host übersteigt, wird sie auf den Maximalwert gesetzt. In jedem Fall wird die Anzahl der tatsächlich gelesenen Wörter zurückgegeben.

Parameter `uint16 addr`: Die Adresse, ab der gelesen werden soll.
`uint16 *data`: Zeiger auf einen Pufferspeicher, der die gelesenen Daten aufnimmt.
`size_t length`: Anzahl der 16-Bit-Wörter, die aus dem EZ-Host gelesen werden sollen.

Rückgabewert `uint16`: Anzahl der gelesenen Wörter.

[a] Hier werden mittels der GPIO-Leitungen des Excalibur die zwei erforderlichen Zugriffszyklen pro Datum koordiniert, s.a. Abschnitt 4.3

hpiReadMailbox()

Überblick Liest die im Mailboxregister gespeicherte Nachricht.

Details In der AHB-Version dieser Funktion wird ein Lesezyklus auf die Mailboxadresse 0xC0C6 durchgeführt, wobei gleichzeitig Adressbit 16 auf 1 gesetzt wird. Dies veranlasst den AHB-Slave, einen entsprechenden Lesezugriff auf das Mailboxregister durchzuführen. In der DPSRAM-Version erfolgt ein *hpiRead* Aufruf auf die Mailboxadresse. *Da die DPSRAM-Versionen der Software im Verlauf der Arbeit nicht weiter verfolgt wurde, besteht die Möglichkeit, dass diese Implementation der Funktion in der letzten Fassung der* ECCL *nicht mehr fehlerfrei ist!*

Parameter keine

Rückgabewert `uint16`: Die im Mailboxregister gespeicherte Nachricht.

hpiReset(state)

Überblick Setzt die Resetleitung des AHB-Slaves auf den übergebenen Wert.

Details Bei Verwendung des AHB-Slaves kann von Excaliburseite ein Reset des EZ-Host ausgelöst werden. Dazu muss die invertierte Resetleitung aktiviert werden. Dies kann, abhängig von der Hardware-Implementation, auf zwei Arten geschehen:

1. Durch Setzen des GPIO-Registers, falls der Reset-Eingang des AHB-Slave-Moduls mit einer der GPIO-Leitungen verbunden wurde.
2. Durch Setzen eines dedizierten Registers im CIAControl-Modul. In diesem Fall wird von *hpiReset* lediglich die in `cialib.h` deklarierte Funktion *setHPIBridgeReset* aufgerufen.

Die Compilerdirektive USE_AHB entscheidet, welche der Varianten verwendet wird.

Parameter `unsigned int state` Gewünschter Zielzustand der Resetleitung.

Rückgabewert `int`: Der neue Zustand der Resetleitung.

hpiStatus()

Überblick Liest das HPI Statusregister aus.

Details Von dieser Funktion existieren zwei Varianten. In beiden werden Bit 16 und 17 der Leseadresse auf 1 gesetzt (`g_hpi_p + 0x18000`)[a] und danach ein regulärer Lesezyklus eingeleitet.

Parameter keine

Rückgabewert `uint16`: Die im Statusregister gespeicherte Nachricht.

[a] `g_hpi_p` ist ein 16-Bit-Zeiger und wird hier um 0x18000 Schritte inkrementiert.

hpiWrite(addr, data, length)

Überblick Schreibt Daten aus einem Puffer in den internen Speicher der EZ-Host-Chips.

Details Diese Funktion ist das Analogon zu *hpiRead*, und zwar sowohl in der AHB-Implementierung als auch in der DPSRAM-Variante. Der maßgebliche Unterschied[a] ist die Richtung der Zuweisung, hier also im Kern: `*(g_hpi_p + addr) = data[counter];`

Parameter `uint16 addr`: Die Adresse, ab der geschrieben werden soll.
`uint16 *data`: Zeiger auf einen Pufferspeicher, aus dem die zu schreibenden Daten ausgelesen werden sollen. Dieser muss zuvor mit Daten gefüllt werden.
`size_t length`: Anzahl der 16-Bit-Wörter, die in den internen Speicher des EZ-Host geschrieben werden sollen.

Rückgabewert `uint16`: Anzahl der geschriebenen Wörter.

[a] In der DPSRAM-Fassung existieren zusätzlich abweichende Steuersignale.

hpiWriteMailbox(message)

Überblick Schreibt eine Nachricht in das EZ-Host Mailboxregister

Details Die AHB-Implementierung dieser Funktion ist vollständig symmetrisch zu *hpiReadMailbox* und besteht aus einer Zeile: `*(g_hpi_p + 0xE063) = message;`. Die DPSRAM-Variante weicht etwas von der Schwesterfunktion ab, hier wird ein eigener Lesezyklus mit gesetztem Adressbit 16 implementiert, statt lediglich *hpiWrite* aufzurufen.

Parameter `uint16 message`: Die zu schreibende Nachricht.

Rückgabewert `uint16`: 0[a]

[a] Hier war eine eventuelle Fehlermeldung als Rückgabewert vorgesehen. Da ein missglückter Schreibzugriff keine Fehlermeldung zurückliefert, wurde diese Funktionalität nie implementiert.

unmapRegions()

Überblick Gibt den gespiegelten Speicherbereich der HPI-Schnittstelle und der Excalibur-Register frei.

Details Diese Funktion ist das Pendant zu *hpiInit()*. Sie sollte am Ende jedes Programmes stehen, das auf die ECCL zugreift. Die von *hpiInit()* allozierten Speicherbereiche, die die gespiegelten Schnittstellen von HPI und Excalibur-Register enthalten, werden wieder freigegeben. *Wird diese Funktion nicht aufgerufen, so besteht die Gefahr eines Speicherlecks!*

Parameter keine

Rückgabewert keiner

A.2 CYMON-Funktionsreferenz

In dieser Referenz werden Funktionen und Datenstrukturen, die innerhalb des CYMON-Quellcodes definiert sind, in `monospace`-Type gesetzt. Systemfunktionen oder ECCL-Aufrufe sind *kursiv*. Aus den von Cypress definierten Headerdateien `lcp_cmd.h` und `lcp_data.h` entnommene Bezeichnungen von Registern und LCP-Kommandos sind serifenlos.

dumpCommand(CBW command)

Überblick Gibt den Inhalt des in `command` gespeicherten Command Block Wrappers auf `stdout` aus.

Details Der Command Block Wrapper (`CBW`) ist eine in [MSBO] spezifizierte Datenstruktur. In Unterabschnitt 6.1.2 findet sich eine umfassende Beschreibung. Diese Funktion wurde zur Überwachung des Laufzeitverhaltens des EZ-Host-Chips entwickelt, könnte aber auch in Richtung Datenquelle für den Massenspeicher erweitert werden. Der Excalibur könnte damit Einblick in die Kommunikation über die USB-Schnittstelle erhalten und ggf. die angeforderten Daten liefern. Geeigneter wäre allerdings eine auf Unterbrechungen basierende Lösung.[a]

Parameter `CBW command`: Das auszugebende Kommando.

Rückgabewert keiner

[a] S. Kapitel 7 für eine Diskussion dieses Themas.

dumpReply(CSW reply)

Überblick Gibt den Inhalt des in `reply` gespeicherten Command Status Wrappers auf `stdout` aus.

Details Diese Funktion ist das Gegenstück zu `dumpCommand`. Die Beschreibung des `CSW` findet sich ebenfalls in Unterabschnitt 6.1.2.

Parameter `CSW reply`: Der auszugebende Statusblock.

Rückgabewert keiner

handleChoice(int selection, char **argv)

Überblick Bearbeitet die mit `selection` gegebene Option.

Details Diese Funktion besteht aus einer großen `switch`-Operation. Der in `selection` gegebene Optionsbuchstabe wird mit den bekannten Optionen verglichen. Ist die Option vorhanden, so wird an die entsprechende Stelle verzweigt. Für manche der Optionen werden Subroutinen ausgeführt. Ist der Optionswert unbekannt, so wird eine Fehlermeldung ausgegeben, gefolgt von einem Aufruf von `printUsage()`.

Parameter `int selection`: Kürzel der aktuell bearbeiteten Option, wird als `char` interpretiert.
`char **argv`: Der `main`-Argumentvektor, enthält die Parameter der aktuellen Option.

Rückgabewert `int`: `errno` falls ein Fehler auftritt, 0 sonst.

init()

Überblick Initialisiert das Programm.

Details Diese Funktion ruft die ECCL-Funktion *hpiInit* auf, um die Bibliothek zu initialisieren. Danach wird dem internen, globalen Zeiger `g_registerBase_p` über die Bibliotheksfunktion *getMappedRegisterBase* die Basisadresse der Excaliburregister zugewiesen, die anschließend zur Reservierung der Unterbrechungsleitung 4 verwendet wird. *Dies ist eine vorbereitende Funktionalität, da in der aktuellen Version der Firmware keine Unterbrechungen des Excalibur-Prozessors unterstützt werden.*

Parameter keine

Rückgabewert `int`: `errno` falls ein Fehler auftritt, 0 sonst.

jumpToAddress(int address)

Überblick Überlädt den Programmzähler des EZ-Host-Prozessorkerns mit der in `address` angegebene Adresse.

Details Diese Funktion schreibt zunächst mittels eines *hpiWrite* Aufrufs die über den `address`-Parameter gegebene Zieladresse in das COMM_CODE_ADDR-Register. Anschließend wird über *hpiWriteMailbox* das LCP-Kommando[a] COMM_JUMP2CODE ausgeführt. Dieses veranlasst den EZ-Host, einen entsprechenden Sprung auszuführen.

Parameter `int address`: Sprungziel als absolute Speicheradresse

Rückgabewert `int`: 0

[a] Link Control Protocol, s. Unterabschnitt 6.2.1.

printUsage(char *toolname)

Überblick Gibt in kurzer Form alle vom CYMON erkannten Optionen und die dazugehörigen Parameter aus.

Details Die Ausgabe erfolgt in der üblichen Schreibweise, d.h. Optionen werden mit ihren Parametern in eckigen Klammern angegeben.

Parameter `char *toolname`: Wird verwendet, um den vollständigen Pfad auszugeben. Dieser Parameter ist üblicherweise das erste Argument der `main`-Routine (`argv[0]`).

Rückgabewert keiner

readData(int address, int length)

Überblick Liest `length` Datenwörter ab der Adresse `address` aus dem Speicher des EZ-Host.

Details Diese Funktion ist im Wesentlichen eine Verpackung für die *hpiRead*-Funktion, übernimmt aber zusätzlich die Aufgabe, einen passenden Datenblock zu allozieren und den Erfolg dieses Speicherzugriffs zu überprüfen.

Parameter `int address`: Die Adresse, ab der gelesen werden soll.
`int length`: Die Anzahl der 16-Bit-Datenwörter, die gelesen werden soll.

Rückgabewert `int`: -1 wenn ein Speicherzugriffsfehler aufgetreten ist, die Anzahl der tatsächlich gelesenen Wörter sonst.

sendMailboxMessage(int message)

Überblick Schreibt die in `message` übergebene Nachricht in das Mailboxregister des EZ-Host.

Details Auch diese Funktion ist nur eine Verpackung um die ECCL-Funktion *hpiWriteMailbox*.

Parameter `int message`: Die zu sendende Nachricht.

Rückgabewert keiner

uploadFile(int address, char* filename)

Überblick Öffnet die in `filename` angegebene Datei, liest den Inhalt und schreibt ihn ab der in `address` angegebenen Stelle in den Speicher des EZ-Host.

Details Die Datei wird mit einem *fopen*-Systemaufruf geöffnet. Tritt hier ein Fehler auf, so gibt die Funktion -1 zurück und beendet sich. Andernfalls wird durch eine Folge von weiteren Systemfunktionen[a] die Größe der Datei bestimmt und ein entsprechender Speicherblock alloziert. Dieser wird mit der ECCL-Funktion *hpiWrite* in die spezifizierte Adresse geschrieben, danach wird der Speicherblock wieder freigegeben. Fehler beim Lesen der Datei oder allozieren des Speichers werden mit Hilfe der *perror*-Funktion ausgegeben.

Parameter `int address`: Adresse, ab der der Dateiinhalt im Speicher abgelegt werden soll.
`char *filename`: Vollständiger Pfad zu der Binärdatei, die geladen werden soll.

Rückgabewert `int`: -1 falls die Datei nicht geöffnet werden konnte, 0 sonst.

[a] *fseek, ftell, rewind.*

writeData(int address, int length)

Überblick Liest `length` Wörter von *stdin* und schreibt sie anschließend ab Adresse `address` in den Speicher des EZ-Host.

Details In dieser Funktion wird zunächst der Speicherbedarf ermittelt und ein passender Block alloziert. Danach werden in einer Schleife mittels *scanf*-Aufrufen die Datenwörter im präfixfreien Hexadezimalformat in den Speicherblock eingelesen. Dieser wird anschließend mit einem einzelnen *hpiWrite*-Aufruf in den Speicher des EZ-Host geschrieben. Hierbei ist zu beachten, dass tatsächlich `length` Wörter vom Anwender eingelesen werden, die *hpiWrite*-Funktion aber die angeforderte Länge überprüft. Es werden nicht mehr Daten geschrieben, als vom Adressraum des EZ-Host abgedeckt werden können. Der Rückgabewert zeigt die tatsächlich geschriebene Zahl an. Mit dieser Funktion können auch über übliche Methoden (pipe) Dateien oder die Ausgabe von anderen Programmen umgeleitet werden, wobei die Eingabe aus präfixfreien, vierstelligen Hexadezimalwerten bestehen muss.[a]

Parameter `int address`: Adresse, ab der geschrieben werden soll.
`int length`: Anzahl der 16-Bit-Datenwörter, die eingelesen und geschrieben werden soll.

Rückgabewert `int`: -1 wenn ein Speicherzugriffsfehler aufgetreten ist, die Anzahl der tatsächlich geschriebenen Wörter sonst.

[a] Die Leseroutine im Quelltext: `scanf( ''%04hx'', &g_data_p[counter]);`

A.3 Quelltexte der entwickelten Software

Die vollständigen Quelltexte der Kommunikations-Bibliothek ECCL, des Monitorprogrammes CYMON und der Firmware für den EZ-Host liegen auf CD-ROM vor.

A.4 In der Firmware verwendeter USB-Deskriptorensatz

Off	Feldname	Gr	Wert	Beschreibung
0	bLength	1	0x12	Länge dieses Deskriptors in Byte
1	bDescriptorType	1	0x01	Geräte-Deskriptor-Konstante
2	bcdUSB	2	0x0110	Binary Coded Decimal, vom Gerät unterstützte USB Version, hier 1.10
4	bDeviceClass	1	0x00	Geräteklasse, wie vom USB-IF definiert. Der Wert 0x00 bedeutet, dass die eigentliche Klasseninformation für jede Schnittstelle im zugehörigen Schnittstellen-Deskriptor angegeben wird.
5	bDeviceSubClass	1	0x00	Geräteunterklasse, wie vom USB-IF definiert. Wenn die Geräteklasse 0x00 ist, so muss auch die Unterklasse 0x00 sein.
6	bDeviceProtocol	1	0x00	Protokollcode, wie vom USB-IF definiert. Ein Wert von 0x00 bedeutet, dass auf Geräteebene kein spezifisches Protokoll verwendet wird. Es können auf Schnittstellenebene Protokollangaben gemacht werden
7	bMaxPacketSize0	1	0x40	Maximale Paketgröße für den EP0 (in Bytes)
8	idVendor	2	0x067b	Vom USB-IF vergebene Hersteller-ID. Die hier verwendete ist von Prolific Technology Inc., einem Hersteller von USB Massenspeicher. Hier sollte für ein fertiges Produkt eine eigene ID beantragt werden
10	idProduct	2	0x2317	Vom Hersteller festgelegte, eindeutige Produkt-ID. Die hier verwendete ID bezeichnet eine USB-Flashdisk des Herstellers Prolific Technology
12	bcdDevice	2	0x0001	Versionsnummer des Gerätes als Binary Coded Decimal

Off	Feldname	Gr	Wert	Beschreibung
14	iManufacturer	1	0x01	Indexnummer des Zeichenketten-Deskriptors, welcher den Hersteller beschreibt, s. Tabelle A.5
15	iProduct	1	0x02	Indexnummer des Zeichenketten-Deskriptors, welcher das Produkt beschreibt
16	iSerialNumber	1	0x03	Indexnummer des Zeichenketten-Deskriptors, welcher die Seriennummer des Produktes enthält
17	bNumConfigurations	1	0x01	Anzahl der vom Gerät unterstützten Konfigurationen, für jede muss ein entsprechender Konfigurations-Deskriptor existieren

Tabelle A.1: Geräte-Deskriptor

Off	Feldname	Gr	Wert	Beschreibung
0	bLength	1	0x09	Länge dieses Deskriptors in Byte
1	bDescriptorType	1	0x02	Konfigurations-Deskriptor-Konstante
2	wTotalLength	2	0x0020	Gesamtlänge dieses und der folgenden Schnittstellen- und Endpunkt-Deskriptoren. Wird z. B. verwendet um den kompletten Deskriptorensatz für eine Konfiguration in einem Transfer abzurufen
4	bNumInterfaces	1	0x01	Anzahl der für diese Konfiguration definierten Schnittstellen
5	bConfigurationValue	1	0x01	Dieser Wert wird als Argument für den *SetConfiguration()* Request verwendet, um diese Konfiguration zu wählen
6	iConfiguration	1	0x00	Index der diese Konfiguration beschreibenden Zeichenkette
7	bmAttributes	1	0xC0	Beschreibt Attribute dieser Konfiguration. Das oberste Bit ist aus historischen Gründen immer gesetzt. Bit 6 legt fest, dass diese Konfiguration des Gerätes keine Energieversorgung über den Bus benötigt. Bit 5 (hier nicht gesetzt) meldet das Gerät als „Remote Wakeup“-fähig. Die Bits 4 bis 0 sind reserviert und immer 0

Off	Feldname	Gr	Wert	Beschreibung
8	bMaxPower	1	0x00	Der maximale, vom Gerät aus dem Bus bezogene Strom, in Einheiten von 2 mA. Nur gültig, wenn Bit 6 des bmAttributes-Feldes nicht gesetzt ist

Tabelle A.2: Konfigurations-Deskriptor

Off	Feldname	Gr	Wert	Beschreibung
0	bLength	1	0x09	Länge dieses Deskriptors in Byte
1	bDescriptorType	1	0x04	Schnittstellen-Deskriptor-Konstante
2	bInterfaceNumber	1	0x00	Laufende Nummer der in dieser Konfiguration definierten Schnittstelle
3	bAlternateSetting	1	0x00	Laufende Nummer einer alternativ zu dieser verwendbaren Schnittstelle innerhalb der aktuellen Konfiguration. Hier identisch zur bInterfaceNumber, es existiert also keine Alternative
4	bNumEndpoints	1	0x02	Anzahl der von dieser Schnittstelle verwendeten Endpunkte (exklusive des stets vorhandenen EP0)
5	bInterfaceClass	1	0x08	Vom USB-IF festgelegte Konstante definiert die Geräteklasse für diese Schnittstelle. 0x08 ist reserviert für Massenspeicher (s. [MSBO])
6	bInterfaceSubClass	1	0x04	Vom USB-IF festgelegte Konstante definiert in der Geräteunterklasse das verwendete Hardwareprotokoll. Der Wert 0x04 steht für das verwendete UFI (s. [MSSO])
7	bInterfaceProtocol	1	0x50	Innerhalb der Geräteklasse und -unterklasse verwendetes Transferprotokoll. Der Wert 0x50 spezifiziert das Mass Storage Bulk Only Protokoll (s. [MSBO])
8	iInterface	1	0x00	Index des Zeichenketten-Deskriptors, der diese Schnittstelle beschreibt

Tabelle A.3: Schnittstellen-Deskriptor

Endpunkt 1 Bulk-Out				
0	bLength	1	0x07	Länge dieses Deskriptors in Byte
1	bDescriptorType	1	0x05	Endpunkt-Deskriptor-Konstante

Off	Feldname	Gr	Wert	Beschreibung
2	bEndpointAddress	1	0x01	Bits 3 bis 0 bestimmen die Nummer des EPs. Die Bits 6 bis 4 sind reserviert und immer 0. Bit 7 bestimmt die Richtung des Endpunktes und ist hier nicht gesetzt, also ist EP1 Outgoing, d.h. Host-to-Device
3	bmAttributes	1	0x02	Bits 1 bis 0 legen den Typ des EPs fest, hier wird Bulk spezifiziert. Bits 5 bis 2 sind nur für Isochronous-EPs von Bedeutung, für andere Typen sind sie auf 0 festgelegt. Bits 7 bis 6 sind reserviert und immer 0
4	wMaxPacketSize	2	0x0040	Die maximale Paketgröße für diesen EP
6	bInterval	1	0x00	Intervall zwischen den Hostabfragen dieses EPs (Polling). Für Bulk-EPs wird dieser Wert ignoriert

Endpunkt 2 Bulk-In				
0	bLength	1	0x07	Länge dieses Deskriptors in Byte
1	bDescriptorType	1	0x05	Endpunkt-Deskriptor-Konstante
2	bEndpointAddress	1	0x82	Endpunkt 2, Ingoing, d.h. Device-to-Host
3	bmAttributes	1	0x02	Bulk-Endpunkt
4	wMaxPacketSize	2	0x0040	s.o.
6	bInterval	1	0x00	s.o.

Tabelle A.4: Endpunkt-Deskriptoren

Herstellername				
0	bLength	1	0x42	Länge dieses Deskriptors in Byte. Hieraus wird die Länge der Zeichenkette berechnet, diese ist nicht Null-terminiert. Die von Cypress vorgegebenen Typendefinitionen geben hier eine Zeichenkettenlänge von 32 16-Bit-Zeichen vor, daher sind die Zeichenketten-Deskriptoren immer 0x42 Byte groß
1	bDescriptorType	1	0x03	Zeichenketten-Deskriptor-Konstante

Off	Feldname	Gr	Wert	Beschreibung
2	bString	0x40	Leading Driver Co.,LTD.	Informationen zum Geräte-hersteller als Zeichenkette in Unicode-Kodierung[1]
Produktname				
0	bLength	1	0x42	Länge dieses Deskriptors in Byte.
1	bDescriptorType	1	0x03	Zeichenketten-Deskriptor-Konstante
2	bString	0x40	USB Mass Storage Device	Informationen zum Produkt als Unicode-Zeichenkette
Geräteseriennummer				
0	bLength	1	0x42	Länge dieses Deskriptors in Byte.
1	bDescriptorType	1	0x03	Zeichenketten-Deskriptor-Konstante
2	bString	0x40	673007E 570001	Eindeutige Geräteseriennummer. [MSBO] fordert hier eine mindestens zwölfstellige Nummer, bestehend aus den Unicodezeichen 0x0030 bis 0x0039 und 0x0041 bis 0x0046, das sind die Ziffern 0-9 und die Buchstaben A-F

Tabelle A.5: Zeichenketten-Deskriptoren

[1] Unicode ist ein Standard, der Schriftzeichen verschiedener Sprachen in 16-Bit-Werten kodiert. In der aktuellen Revision 4.0 umfasst Unicode 96.382 Zeichen. Für umfassende Dokumentation s. `http://www.unicode.org`.

Anhang B

Graphiken und Diagramme

B.1 AHB-Slave

B.1.1 Simulationsergebnisse

Der AHB-Slave wurde in der softwarebasierten Simulationsumgebung ModelSim simuliert. Abbildung B.1 zeigt die Simulationsergebnisse. Getestet wurde zuerst ein Schreibzugriff auf die Adresse 0x1000, geschrieben wurde der Datenwert 0xBEEF. Der Transfer ist zwischen 300 ns und 640 ns sichtbar. Der zweite Test ist ein Zugriff auf die nächsthöhere Adresse 0x1002 (640 ns – 940 ns). Aus dem Signal HWDATA wird ersichtlich, dass die an den EZ-Host über den HPI_IO-Datenbus weitergeleiteten Daten in der Tat den richtigen, oberen 16 Bit des Datenwortes entsprechen. Es folgen je ein Lesezugriff auf das Mailboxregister (940 ns – 1110 ns) und das Statusregister (1110 ns – 1290 ns). Da der EZ-Host nicht simuliert wird, erfolgt keine Antwort. Der letzte gültige Transfer ist ein Schreibzugriff auf das Mailboxregister, es wird das LCP-Kommando COMM_EXEC_INT geschrieben, das entspricht dem Datenwort 0xCE01. Im Anschluss wird noch ein Leseversuch auf ein 32-Bit-Wort durchgeführt. Die statische Fehlerantwort des AHB-Slave, 0xDEADFACE, ist auf dem AHB-Lesedatenbus HRDATA sichtbar.

B.1.2 Mitschnitte aus Logikanalysator-Messungen

Da an der CIA-Karte nur 10 Pins für den Anschluss des Logikanalysators zur Verfügung standen, sind nur die unteren 4 Bit des Datenbusses dargestellt, sie zeigen die jeweils unterste Stelle des 16-Bit-Wertes in Hexadezimaldarstellung. Die anderen dargestellten Signale sind der 2 Bit breite Adressbus (in Binärdarstellung), die low-aktiven Read-Enable-, Write-Enable- und Chip-Select-Signale sowie das Unterbrechungssignal. Abbildung B.2 zeigt die vier getesteten Zugriffe auf den EZ-Host über den AHB-Slave.

Lesezugriff

Gelesen wurde der zuvor geschriebene Wert 0x1234 von der Adresse 0x1000. Die fallende Flanke des WR_N-Signals ca. 50 ns nach aktiv werden von CS_N liegt mitten in der HPI Address-Phase des ADDR-Signals. Mit der steigenden Flanke wird die Adresse (0 als unterste Hexadezimalstelle des realen Wertes 0x1000) übernommen. Danach folgt der Lesezyklus. Der Adressbus zeigt HPI Data, der gelesene Datenwert ist die erwartete 4.

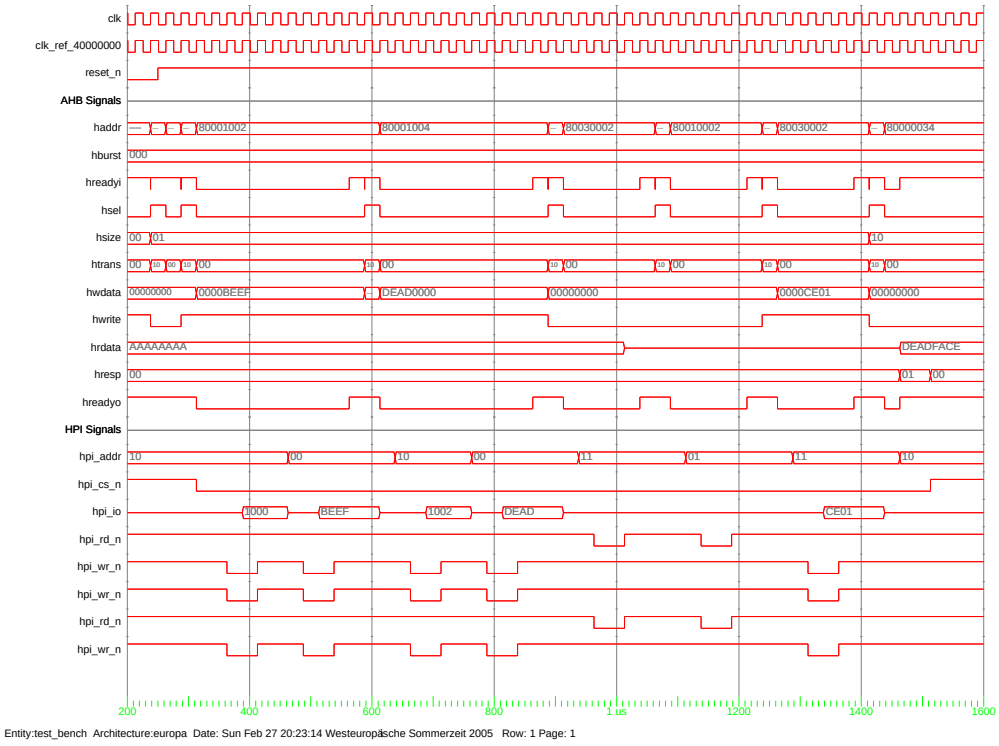


Abbildung B.1: Ergebnis der Simulation des AHB-Slaves

Schreibzugriff

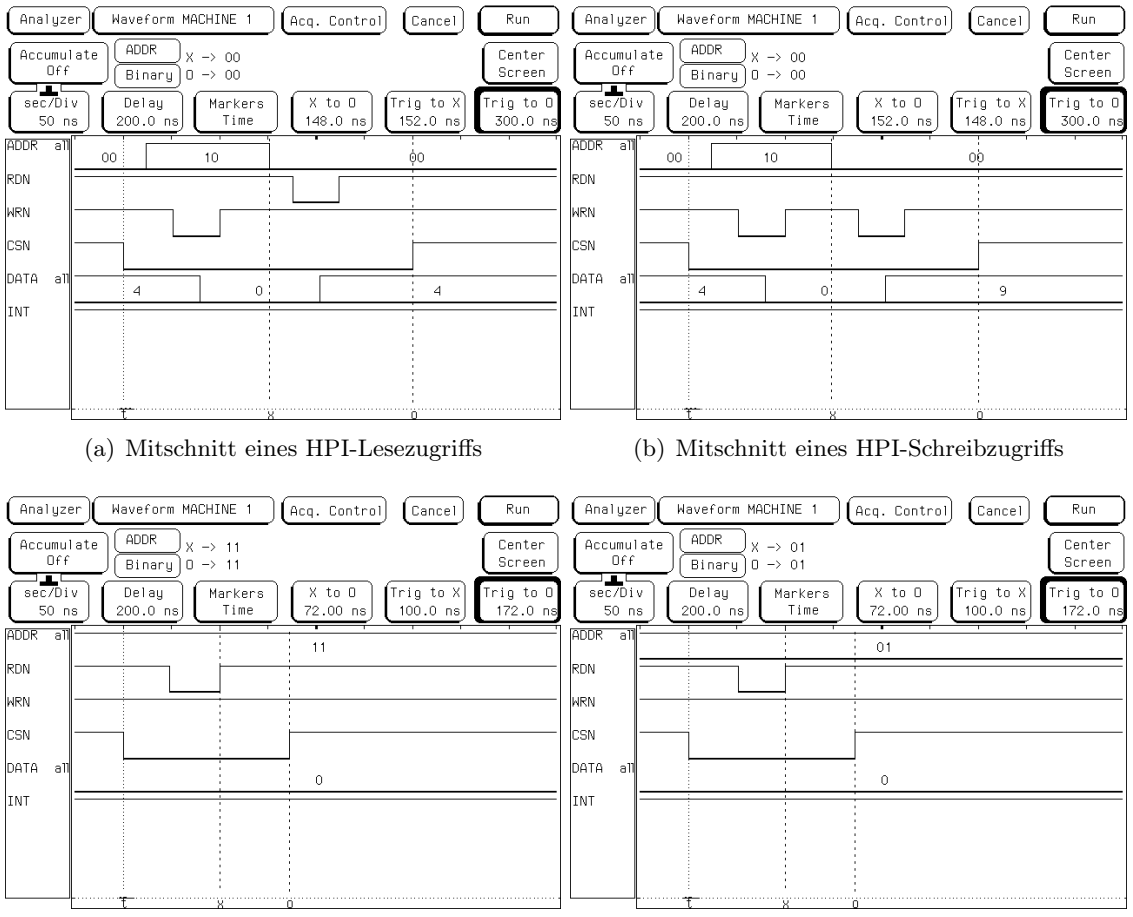
Hier wurde der Wert 0x5678 an die Adresse 0x1000 geschrieben. Der erste Zyklus ist wegen der gleichen Adresse identisch und stimmt auch in den Signalen überein. Der zweite Schreibzyklus schließt sich an. Der Adressbus zeigt wieder HPI Data, der angezeigte Datenwert ist die 9.

Lesezugriff auf das Statusregister

Gelesen wurde der Wert 0x0030, daher zeigt der Datenbus keine Änderung der Aktivität. Man erkennt jedoch den mit ca. 175 ns deutlich kürzeren, einfachen Zyklus sowie das HPI Status-Signal auf dem Adressbus.

Lesezugriff auf das Mailboxregister

Auch hier ist das HPI Mailbox-Signal deutlich zu erkennen. Das Mailboxregister enthielt den Startwert 0x0000. Auf die Darstellung eines Schreibzugriffs wurde verzichtet.



(a) Mitschnitt eines HPI-Lesezugriffs

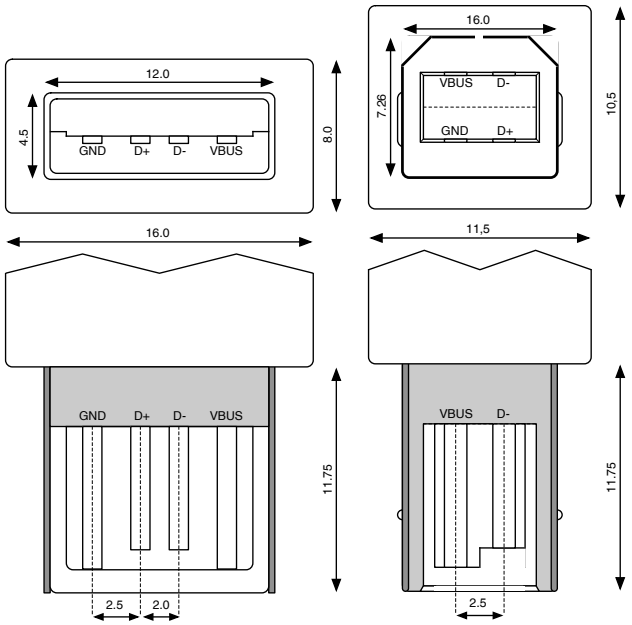
(b) Mitschnitt eines HPI-Schreibzugriffs

(c) Mitschnitt eines HPI-Statusregisterzugriffs

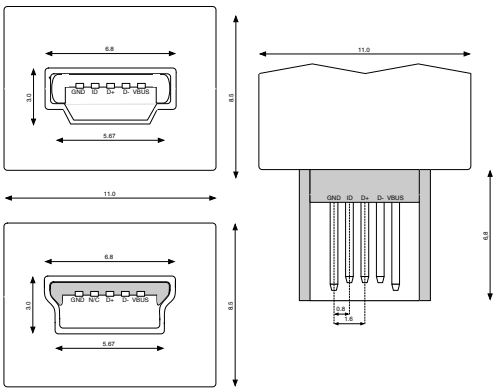
(d) Mitschnitt eines HPI-Mailboxregisterzugriffs

Abbildung B.2: Mit dem Logikanalysator gemessene HPI-Zyklen. (a) Lesezugriff, (b) Schreibzugriff, (c) Auslesen des Statusregisters, (d) Auslesen des Mailboxregisters.

B.2 USB-Steckertypen



(a) USB-Typ-A (b) USB-Typ-B



(c) USB-Typ-Mini-A und Mini-B

Abbildung B.3: USB-Stecker (a) Typ-A, (b) Typ-B, (c) Typ-Mini-A und Mini-B

Danksagung

Mein erster Dank gilt Herrn Prof. Dr. Volker Lindenstruth, der mir die Möglichkeit gegeben hat, mich in einem interessanten und vielseitigen Arbeitsgebiet zu bewähren.

Des Weiteren bedanke ich mich bei allen Angehörigen des Lehrstuhls für Technische Informatik, die mir bei Fragen und Verständnisproblemen stets mit Rat und Tat zur Seite standen. Die freundschaftliche und anregende Atmosphäre hat die Mitarbeit zu einem bemerkenswerten Erlebnis werden lassen.

Besonders hervorheben möchte ich Herrn Ralf Panse für seine Hilfe im Umgang mit den diversen Soft- und Hardwarewerkzeugen, die zur Fertigstellung meiner Arbeit notwendig waren. (Linux läuft!)

Danken möchte ich auch Herrn Jan de Cuveland, dessen zahllose hilfreiche Hinweise im Umgang mit LaTeX mir halfen, die Klippen des Textsatzes erfolgreich zu umschiffen.

Den Herren Rolf Schneider und Christian Reichling will ich dafür danken, dass sie immer mit einem aufmunternden Wort zur Stelle waren, wenn mir der Schwung ausging. (Sieben Tomaten! Matchball!)

Herrn Holger Höbbel danke ich für die Anfertigung der Mezzanine-Karte, die mir beim Kennenlernen des EZ-Host-Chips sehr geholfen hat.

Mein letzter, aber sicher nicht geringster Dank gilt meiner Frau Judith Natho, die gerade in der letzten, heißen Phase der Arbeit besonders viel Zeit und Verständnis für mich aufbrachte.

Diese Arbeit ist in aufrichtiger Zuneigung meinem Vater gewidmet, der mich stets ermutigt und unterstützt hat auf meinem oftmals steinigen Weg durch das Studium der Physik, und meinem Physiklehrer, Herrn Dr. Thilo Koch, dessen Unterricht mich bewegt hat, diesen Weg einzuschlagen.

Literaturverzeichnis

[T500] Top 500 Supercomputer List, Nov. 2004, http://www.top500.org

[Pfi98] In Search of Clusters, Gregory Pfister, Pearson Education, 2nd edition Dec. 12, 1997, ISBN 0138997098

[USB20] Universal Serial Bus Specification, Rev. 2.0, April 27, 2000, http://www.usb.org

[MSSO] Universal Serial Bus Mass Storage Class Specification Overview, Rev. 1.2, June 23, 2003, http://www.usb.org

[MSBO] Universal Serial Bus Mass Storage Class Bulk-Only Transport, Rev. 1.0, Sept. 31, 1999, http://www.usb.org

[UFI] Universal Serial Bus Mass Storage Class UFI Command Specification, Rev. 1.0, Dec. 14, 1998, http://www.usb.org

[EHRM] Altera EPXA1 Development Board Hardware Reference Manual, Version 1.1, Sept. 11, 2002, http://www.altera.com

[EDRM] Altera Excalibur Devices Hardware Reference Manual, Version 3.1, Nov. 2002, http://www.altera.com

[AL173] Altera Application Note 173: Dual ported SRAM Reference Design, http://www.altera.com

[CEDS] Cypress EZ-Host Programmable Embedded USB Host/Peripheral Controller Data Sheet, Rev. *E, Oct. 1, 2003, http://www.cypress.com

[CBUM] Cypress BIOS User's Manual, Version 1.1, 2003, http://www.cypress.com

[CFRG] Cypress Frameworks Reference Guide for the CY7C67300/CY7C67200 Family of Products, Version 1.0, 2003, http://www.cypress.com

[CY16] Cypress CY16 USB Host/Slave Controller/16-Bit RISC Processor Programmers Guide, Version 1.1, Feb. 7, 2003, http://www.cypress.com

[AARM] ARM Architecture Reference Manual, Issue E, June 2000, http://www.arm.com

[AMBA] ARM AMBA^{TM}Specification (Rev. 2.0), Issue A, May 13, 1999, http://www.arm.com

[CREF] C kurz & gut, Peter Prinz & Ulla Kirch-Prinz, O'Reilly Verlag, 1. Auflage 2002, ISBN 3897212382

Hinweis: Einige Hersteller untersagen so genannte „Deep Links" auf innere Strukturen ihrer Internetpräsenz, daher verweisen Links in dieser Arbeit grundsätzlich auf die Startseite des jeweiligen Herstellers.

Erklärung zur selbständigen Verfassung

Ich versichere, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 1. März 2005

Georg Klein